Hochschule Offenburg

Fakultät Medien und Informationswesen
Studiengang Medien und Informationswesen

Bachelorarbeit

„Startbeschleunigung durch lokales Caching von Geschäftsobjekten eines Adservers “

Michael Behnke

Wintersemester 2014/2015

Bearbeitungszeitraum

01. 09. 2014 – 31. 12. 2014

Betreuer

Prof. Dr. Volker Sänger

Birger Brunswiek

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Offenburg, den 15.12.2014 _____

Unterschrift

Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die mich während der Arbeit an meiner Bachelor-Thesis unterstützt haben.

Insbesondere gilt mein Dank Birger Brunswiek, der mich durch viele Anregungen und kritisches Hinterfragen meiner Lösungsansätze auf ständig neue Wege brachte, und Prof. Dr. Volker Sänger, ohne dessen Hilfestellung und Erfahrung diese Arbeit sicher nicht möglich gewesen wäre.

Des Weiteren möchte ich mich beim Team des Backends bedanken, die bei Fragen jederzeit zur Verfügung standen. Wann immer ich an einem Problem festhing, kam die entscheidende Idee zur Lösung von ihnen. Auch danke ich der ADITION technologies AG dafür, dass sie mir die Möglichkeiten gegeben haben, die Bachelor-Thesis in ihrem Unternehmen zu schreiben, um mich selbst zu beweisen und weiterzuentwickeln.

Nicht zuletzt gebührt meinen Eltern sowie meinen Geschwistern Dank, die in dieser Zeit immer für moralische und emotionale Unterstützung sorgten.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Über das Unternehmen 1
1.2. Begriffsdefinition Adserver 1
1.3. Problembeschreibung 2
1.4. Aufgabenstellung 3

2. Problemanalyse **5**

2.1. Entwicklung der Startzeiten 5
2.1.1. Fazit der historischen Analyse 8
2.2. Mögliche Ursachen 9
2.3. Ursachenanalyse 10
2.3.1. Datenbankserver-seitige Analyse 11
2.3.2. Client-seitige Analyse 16
2.4. Fazit 18

3. Lösungsalternativen **21**

3.1. Vermeiden von Neustarts 22
3.2. Einsatz verteilter Datenbanken 23
3.2.1. Transparente Verwaltung der Daten 25
3.2.2. Verlässlicher Datenzugriff über verteilte Transaktionen 26
3.2.3. Verbesserte Performance 27
3.2.4. Einfache Systemerweiterung 27
3.2.5. Aufwand für Implementierung 27
3.2.6. Fazit zum Einsatz verteilter Datenbanken 28
3.3. Lokales Caching der Objekte 28
3.3.1. Serialisierung in Datei 29
3.3.2. Serialisierung in den Arbeitspeicher 30
3.4. Restrukturierung der SQL-Statements 32

3.5. Fazit und Auswahl der Methode 33

4. Konzeption **35**
4.1. Besonderheiten Shared Memory 35
4.1.1. Serialisierung mittels Apache Thrift 39
4.2. Prototyp 41
4.2.1. Vermeidung von Objektkopien 45
4.3. Klassendiagramm 47
4.4. Sequenzdiagramm 49
4.5. Testdesign 53

5. Implementierung **55**
5.1. Thrift-Definition 55
5.2. Implementierung als Skelett hinzufügen 56
5.3. Test-Implementierung 56
5.4. Änderungen im Adserver 57

6. Auswertung **61**
6.1. Vorher-/Nachher-Vergleich 61
6.2. Optimierung der Objekterstellung 62
6.3. Fazit der Auswertung 64

7. Ausblick **65**
7.1. Konsistenzsicherstellung der Objekte 65
7.2. Optimierung der Objekterstellung 67
7.3. Optimierung der Datenbankabfragen 68
7.4. Optimierung der Kommunikation mit dem XML-Daemon 69
7.5. Fazit zum Ausblick 70

Zusammenfassung **71**

A. Anhang **73**
A.1. Thrift-generierte Klasse „PersonThrift“ 73
A.2. Prototyp 74

Literaturverzeichnis **77**

1. Einleitung

1.1. Über das Unternehmen

Die ADITION technologies AG, nachfolgend nur ADITION genannt, mit Hauptsitz in Freiburg ist eine Tochter der virtual minds AG und setzt ihren Fokus auf den Vertrieb und die Entwicklung der unabhängigen Adservertechnologie ADITION adserving. Die Basistechnologie wurde bereits 2001 konzipiert, und 2004 in die neu gegründete ADITION technologies AG überführt, die seitdem die Produktentwicklung und Kommerzialisierung weiter vorantreibt. ADITION ist nach DoubleClick (Google) der zweitgrößte AdServing Anbieter im deutschsprachigen Raum, und damit auch einer der wichtigsten Akteure im AdServing-Bereich im gesamten europäischen Markt.

1.2. Begriffsdefinition Adserver

Ein Adserver ist ein Instrument zur Verwaltung von Online-Werbung, und kann zur Auslieferung und Erfolgsmessung dieser verwendet werden (Busch, 2014). Im Grunde genommen handelt es sich um einen HTTP-Server. Benutzer besuchen Websites auf denen Platzhalter für Werbung hinterlegt sind. Werden diese Seiten aufgerufen, sorgen die Platzhalter dafür, dass HTTP-Requests an den Adserver versendet werden. Dieser ermittelt über Informationen, die entweder direkt über den HTTP-Requests mitgeliefert worden sind oder z. B. über Cookie-Daten ermittelt werden können, passende Werbung, und ersetzt mit dieser die Platzhalter. Ein HTTP-Request der an einen Adserver versendet wird, wird auch als Adrequest bezeichnet.

Da es einem einzigen Adserver nicht möglich wäre, die Milliarden monatlich anfallender Adrequests allein zu bewältigen, hat ADITION über 60 dieser Server im

Einsatz. Vorgeschaltete Loadbalancer sorgen dafür, dass die Requests sinnvoll auf die Adserver verteilt werden. Des Weiteren ist jeder Adserver einer Farm zugeordnet (siehe Abbildung 1.1). Durch dieses Vorgehen können Kundengruppen voneinander isoliert werden. Zwar greifen alle Adserver auf die gleiche zentrale Datenbank zu, es werden aber nur die Daten geladen, die zur jeweiligen Farm gehören. Werden daher die Daten durch einen Kunden so verändert, dass eine Überlastung des Adserver entsteht, sind nur die Adserver der jeweiligen Farm betroffen.

1.3. Problembeschreibung

Einer der Schlüsselfaktoren für die Bestimmung der Qualität eines Adservers ist die Performance. Die Webseiten-Betreiber erwarten beim Aufruf ihrer Website die instantane Anzeige von Inhalt, wozu auch Werbebanner zu zählen sind. Dies macht eine Auswertung der passenden Werbung im Mikro- bis Millisekunden Bereich nötig. Als Konsequenz werden die meisten benötigten Daten bereits mit Start eines Adservers aus einer zentralen Datenbank (siehe Abbildung 1.1) in den Arbeitsspeicher geladen, da ein Zugriff auf die Datenbank zur Laufzeit zu lange dauern würde. Dieses Vorgehen sorgt somit für eine allgemein höhere Performance bei der Beantwortung von Adrequests, erhöht aber auf der anderen Seite auch die Zeit, die für den Start des Adservers selbst benötigt wird.

Die Dauer des Starts ist schon isoliert je Adserver betrachtet ein Problem, kumuliert für über 60 Adserver wird es aber zunehmend kritischer. Wird der Programmcode nun aktualisiert, sei es auf Basis geplanter Software-Updates oder zum Einspielen von Hotfixes aufgrund identifizierter Probleme, werden alle Instanzen des Adservers neu gestartet. Je Adserver wird von einem Zeitaufwand für den Neustart von 10 bis 30 Minuten ausgegangen. Geht man davon aus, das diese sequentiell neu gestartet werden - was nicht zwangsläufig der Fall ist - dann würden etliche Stunden benötigt werden, bis alle Adserver in der aktuellsten Version wieder live sind. Setzt man hingegen einen parallelen Neustart der Server voraus ist zu erwarten, dass diese sich gegenseitig blockieren, da alle auf denselben zentralen Datenbankserver zugreifen, und über dasselbe Netzwerk, mit endlicher Kapazität, kommunizieren. Letztlich findet also ein großer Teil des Datenaustauschs seriell statt, und wie man das Problem auch betrachtet, die Startzeit erweist sich als Problem. Der einzuplanende Aufwand für Software-Wartung erhöht sich, und damit einhergehend verringert sich die Zeit

für Neuentwicklung. Es entstehen Kosten.

Insofern ist das zu formulierende Ziel klar: Die Startzeiten für einen Adserver müssen deutlich reduziert werden.

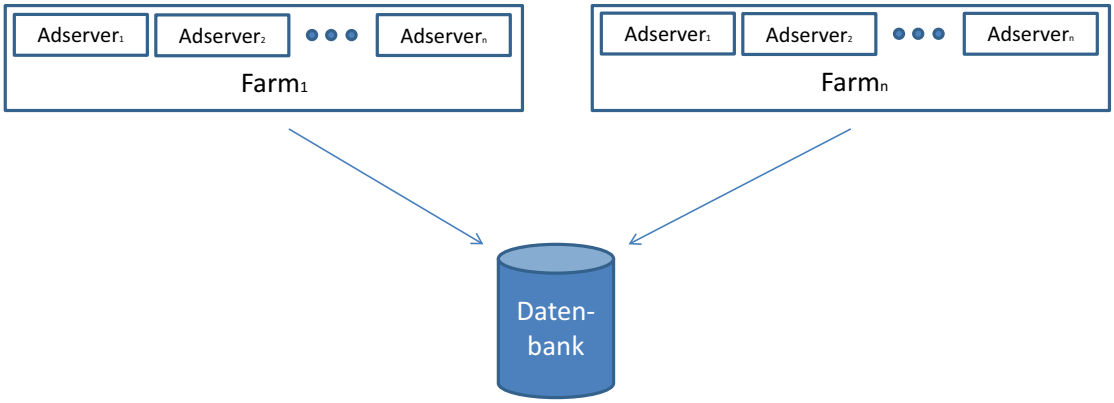

Abbildung 1.1.: Alle Adserver beziehen ihre Daten von der selben Datenbank

1.4. Aufgabenstellung

Die Realisierung des Ziels führt zu folgenden Aufgaben:

1. Die Problemstellung soll analysiert werden, um die Ursachen zu ermitteln. Damit einher geht eine historische Analyse der Entwicklung der Startzeiten.
2. Es sollen Lösungsalternativen aufgelistet und hinsichtlich ihrer Eignung bewertet werden.
3. Eine oder mehrere dieser Alternativen sollen konzipiert, implementiert und getestet werden.
4. Der tatsächliche Erfolg soll gemessen werden.
5. Ein Ausblick auf weitere Optimierungsansätze soll gegeben werden.

Diese Aufgaben werden in den nächsten Kapiteln umgesetzt.

2. Problemanalyse

Bevor passende Lösungsansätze entwickelt werden können, ist die Frage zu klären, welche Ursachen überhaupt zum Problem führen. Diese Frage soll in diesem Kapitel im Detail geklärt werden.

2.1. Entwicklung der Startzeiten

Bis zu einem gewissen Grad ergibt es sogar Sinn einen Schritt früher anzusetzen: Existiert das angesprochene Problem wirklich, und wenn ja, wie stark ist es ausgeprägt?

Die Frage lässt sich beantworten, wenn die Entwicklung der Startzeiten der letzten paar Jahre aller Adserver bis heute analysiert wird. Zu diesem Zweck können Log-Dateien eingesehen werden, die jeder Adserver standardmäßig führt. In einem solchen Log werden nicht nur Fehler und Warnungen ausgegeben, sondern auch spezielle Ereignisse, die während Betrieb und Startphase auftreten. Insbesondere letzteres ist für die reine Analyse der Startdauer von Interesse. Die Daten reichen bis zum März 2012 zurück, wobei der auswertbare Zeitraum je nach Adserver schwankt. Folglich ist eine Auswertung über einen signifikant langen Zeitraum möglich.

Neben der Gesamt-Startdauer werden noch drei weitere Angaben getätigt, die für eine detaillierte Betrachtung des Problems von Belang sind. Effektiv lässt sich damit die Startphase in die drei wichtigsten Teilelemente zerlegen, was eine Betrachtung darüber ermöglicht, welcher davon den größten Einfluss auf die lange Startdauer hat. Dazu zählen:

1. Die Dauer die benötigt wird, um Filter zu laden: Wie bereits in der Erklärung zum Begriff „Adserver“ beschrieben, ermittelt dieser die passende Werbung für jeden Benutzer. Damit Internet-Werbung nur bestimmten Personenkreisen

angezeigt wird, können Werbekampagnen, Werbeplätze und Werbebanner mit Filtern assoziiert werden. Diese filtern die Benutzer auf Basis ihnen zugehöriger Daten, so das letztlich jeder von ihnen nur das sieht, was für ihn bestimmt ist.

2. Die Dauer die benötigt wird, um alle Banner zu laden: Ein Werbebanner ist, aus Kundensicht, das Bild, das dem Benutzer zu einem Artikel oder einer Firma letztlich angezeigt wird, und auf das er klicken kann, um auf eine andere Webseite weitergeleitet zu werden.

3. Die Dauer die benötigt wird, um alle Werbeplätze zu laden. Eine einzelne Webseite kann mehrerer solcher Werbeplätze haben, auf denen unterschiedliche Werbung angezeigt wird.

Ein weiteres Teilelement des Adserver-Starts ist das Laden der Werbekampagnen. Diese werden aktuell aber asynchron geladen, und stehen somit erst nach dem eigentlichen Start zur Verfügung. Der Grund für dieses Vorgehen ist der, dass auch das Laden der Kampagnen sehr lange dauert, und somit der Start noch zeitintensiver werden würde, wenn diese ebenfalls synchron geladen werden. Es ist zwar ein Ziel, den Start möglichst so effizient zu gestalten, dass auch Kampagnen ohne Probleme synchron geladen werden können, aber dies ist vorerst von nachrangiger Bedeutung, und kann daher vernachlässigt werden.

Nachdem identifiziert ist, welche relevanten Informationen in den Log-Dateien vorhanden sind, können diese extrahiert werden, um sie anschließend mittels Tabellenkalkulationssoftware aufzubereiten. Mehrere Aussagen können so aus den Daten ermittelt werden[1]:

1. Die Entwicklung der Startdauer der Adserver seit August 2012 über alle Farmen hinweg bzw. für jeden einzelnen Adserver und jede Farm im Monatsmittel: Die Entwicklung der Startzeiten ist Abbildung 2.1 zu entnehmen. Aktuell liegt die Startdauer gemittelt über alle Adserver aller Farmen bei rund 400 Sekunden. Der Mittelwert wird allerdings durch einige Adserver mit sehr niedriger Startzeit nach unten korrigiert. Für einige Adserver liegt der Wert aktuell bei 900 Sekunden. Des Weiteren ist als kritisch zu betrachten, dass in den letzten Monaten die Startdauer stärker zunimmt als zuvor, was für eine gesteigerte Verschlechterung in den nächsten Jahren spricht.

[1] Es sind noch weitere Informationen extrahierbar, die aber erst zu einem späteren Zeitpunkt eine Rolle spielen

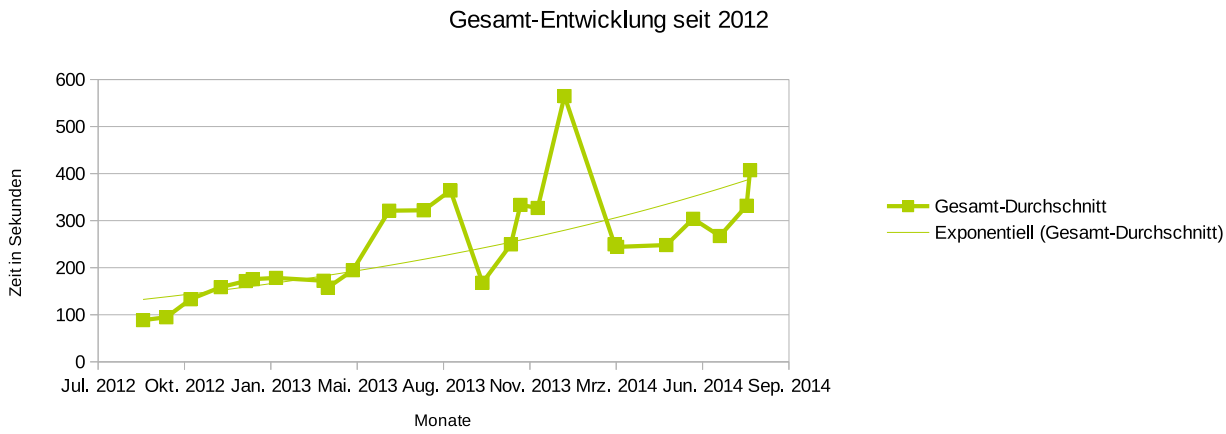

Abbildung 2.1.: Entwicklung der Startzeiten seit Ende 2012

1. Eine Trendlinie, die basierend auf den vergangenen Daten Vorhersagen für die zukünftige Entwicklung treffen kann: Wird die Entwicklung für die nächsten paar Jahre extrapoliert, dann lässt sich darauf schließen, dass der Schnitt der Startdauer bis 2018 auf über 700 Sekunden steigt. Im Januar 2014 war ein starker Anstieg zu vermerken, der auf einen Programmierfehler zurückzuführen war. Nachdem dieser behoben wurde normalisierte sich die Startzeit dann wieder (siehe ebenfalls Abbildung 2.1).

2. Eine Gegenüberstellung der Gesamt-Startdauer seit August 2012 für alle Farmen gemittelt zur Gesamt-Startdauer ohne Berücksichtigung gleichzeitiger Starts. Hintergrund dieser Messung ist die Vermutung, dass gleichzeitige Starts mehrerer Adserver die Startdauer insgesamt größer werden lassen, da alle auf denselben Datenbankserver zurückgreifen. Diese Aussage lässt sich mit der angesprochenen Methode belegen oder widerlegen: Die Ergebnisse sind Abbildung 2.2 zu entnehmen. Der Fokus liegt hier nur auf den fünf wichtigsten Farmen, was dazu führt, dass die Durchschnittszeiten im Allgemeinen ansteigen. Wie zu erwarten ist zu erkennen, dass die Startzeit insgesamt höher ausfällt, wenn mehrere Adserver gleichzeitig starten. Dieser Unterschied ist zwar vorhanden, aber nicht extrem. Das liegt unter anderem daran, dass von den über 60 vorhandenen Adservern selten mehr als fünf bis sieben gleichzeitig starten.

3. Die Pearson-Korrelation zur Messung der linearen Abhängigkeit zweier Variablen. Als Beispiel dienen z. B. die Variablen Körpergröße und Gewicht. Für gewöhnlich ist davon auszugehen, dass größere Menschen mehr wiegen. Die Be-

rechnung der Pearson-Korrelation kann diesen Zusammenhang mathematisch verifizieren oder falsifizieren (Downey, 2012). In Kontext der Problemstellung kann die Berechnung dazu genutzt werden, festzustellen welche Teildauer den größten Einfluss auf die Gesamtdauer hat. Somit lässt sich identifizieren, wo ein Problem zu vermuten ist. Dafür werden folgende Pearson-Korrelation berechnet:

a) Gesamt-Startdauer und Filter-Ladedauer

b) Gesamt-Startdauer und Banner-Ladedauer

c) Gesamt-Startdauer und Banner-Anzahl

d) Gesamt-Startdauer und Werbeplatz-Ladedauer

Die Ergebnisse der Pearson-Korrelation sind Tabelle 2.1 zu entnehmen. Während zwischen der Dauer für das Laden der Banner und der Gesamtdauer annähernd ein Korrelations-Koeffizient von Eins ermittelt wurde, was eine direkte Korrelation indiziert, existiert diese nicht zwischen der Menge geladener Banner und der Gesamtdauer[2]. Dies liegt den Schluss nahe, das sich nicht die Anzahl der Banner in den letzten Jahren verändert hat, sondern deren Größe, so dass insgesamt mehr Daten geladen werden müssen. Bei dieser Aussage ist aber Vorsicht walten zu lassen. Obwohl die Korrelation eine Beziehung zwischen zwei Variablen indiziert, kann faktisch damit nicht ausgesagt werden, ob sich beide negativ oder positiv beeinflussen, oder durch eine weitere unbekannte Variable beeinflusst werden (Downey, 2012). Da Banner ihrerseits wieder Unterobjekte enthalten, z. B. ihnen zugeordnete Filter, die ebenfalls geladen werden müssen, ist es auch möglich, dass sich nichts an den Bannern und deren Größe geändert hat, sondern an der Menge und Größe der Unterobjekte. Die Kausalität hinter der Korrelation kann also eine anderen sein. Nichtsdestoweniger ergibt die Auswertung der Logs, dass 85 % der Zeit die für den gesamten Start aufgewendet werden, auf das Laden der Banner entfällt. Es ist also sicher zu sagen, dass diese näher betrachtet werden sollten, da hier Optimierungspotential zu vermuten ist.

2.1.1. Fazit der historischen Analyse

Die Startdauer beträgt je nach Adserver und Farm aktuell 400 bis 900 Sekunden, mit steigender Tendenz. Dieser Wert ist unhaltbar lange. Als zentrales Problem ist

[2]Der gemessen Wert von 0,4 beschreibt einen vernachlässigbaren schwachen Zusammenhang

	Filter-Ladedauer	Banner-Ladedauer	Banner-Anzahl	Werbeplatz-Ladedauer
r	$< 0{,}2$	~ 1	$\sim 0{,}4$	$< 0{,}2$

Tabelle 2.1.: Pearson-Korrelation (r) zwischen den genannten Variablen und der Gesamt-Startdauer

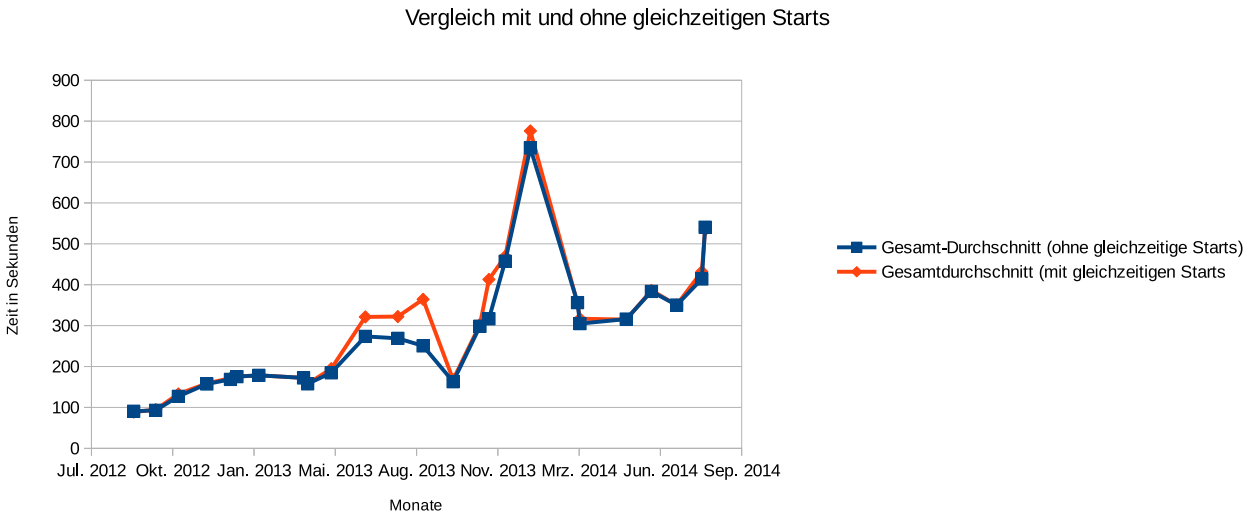

Abbildung 2.2.: Gegenüberstellung der Startzeiten mit und ohne gleichzeitige Adserver-Starts.

das Laden der Banner identifiziert worden, wobei noch unbekannt ist, warum diese so viel Zeit in Anspruch nehmen.

2.2. Mögliche Ursachen

Das beschriebene Problem der langsamen Adserver-Starts kann unterschiedlichste Ursachen haben, wobei einige davon einen unterschiedlichen Lösungsansatz verlangen. Da es für den Projekterfolg nachteilig sein kann, sich bereits anfangs auf bestimmte Bereiche zu limitieren, sind hier all die möglichen Ursachen aufgelistet und kurz beschrieben, die in Frage kommen. Zu beachten ist, dass durchaus die Möglichkeit besteht, das keine der hier genannten Gründe als „Hauptschuldiger“ zu benennen ist. Eine Kombination diverser Faktoren ist ebenso denkbar. Diesem Umstand soll bei der Analyse Beachtung geschenkt werden. Folgende Ursachen kommen in Frage, wobei keine besondere Reihenfolge oder Gewichtung zu Grunde liegt. Die Liste muss nicht zwangsläufig vollständig sein, sollte aber alle zentralen Punkte nen-

nen:

1. Überlastetes Netzwerk: Die Kommunikation jeder der über 60 Adserver erfolgt über dasselbe Netzwerk. Da diesem nur eine endliche Kapazität zur Verfügung steht, ist davon auszugehen, dass insbesondere der gleichzeitige Start mehrerer Adserver zum Problem führt.

2. Überlastete Hardware: Veraltete Hardware kann dazu führen, dass z. B. die CPU nicht mit der Berechnung hinterkommt. Hardware-seitige Engpässe lassen sich durch Profiling relativ leicht identifizieren. Unter Profiling sind Programmierwerkzeuge zu verstehen, die das Verhalten von Programmen analysieren, z. B. hinsichtlich der CPU-Auslastung. Profiling kann dazu genutzt werden, Programmschachstellen aufzudecken, und zu beheben.

3. Langsame SQL-Abfragen: Hier stellt sich die Frage, ob der formulare Aufbau der SQL-Abfragen sinnig ist, und wie viele Abfragen überhaupt gestellt werden.

4. Langsamer Adserver-Code: Ist es womöglich gar nicht die Kommunikation mit dem Datenbank-Server, die zum Problem führt, sondern vielmehr die Verarbeitung der Objekte im Adserver selber. Zur Auswertung dieser Fragestellung kann eine client-seitige Analyse des Problems durchgeführt werden.

2.3. Ursachenanalyse

Die vier genannten möglichen Ursachen lassen sich grundsätzlich in zwei Auswertungsverfahren gruppieren. Ob eine Überlastung des Netzwerks oder der Hardware seitens des Datenbank-Servers vorliegt, lässt sich über Profiling des Datenbank-Servers ermitteln. Dem gegenüber steht langsamer Adserver-Code. Hier lässt sich eine Client-seitige Analyse durchführen, wobei der Adserver in diesem Szenario als Client aufzufassen ist, der die Dienste des Datenbankservers aufruft. Die SQL-Abfragen lassen sich über beide Verfahren evaluieren. Zwar werden diese serverseitig ausgeführt, im Client lässt sich deren Struktur aber besser analysieren und deren Dauer einfacher messen.

2.3.1. Datenbankserver-seitige Analyse

Da die bisherige Vermutung die ist, dass eine Kombination aus einem überlasteten Netzwerk und zu vieler Eingabe/Ausgabe-Operationen auf dem Datenbank-Server als Engpass auftritt, ist es sinnig zuerst auch den Datenbank-Server selbst zu profilen.

2.3.1.1. Testaufbau

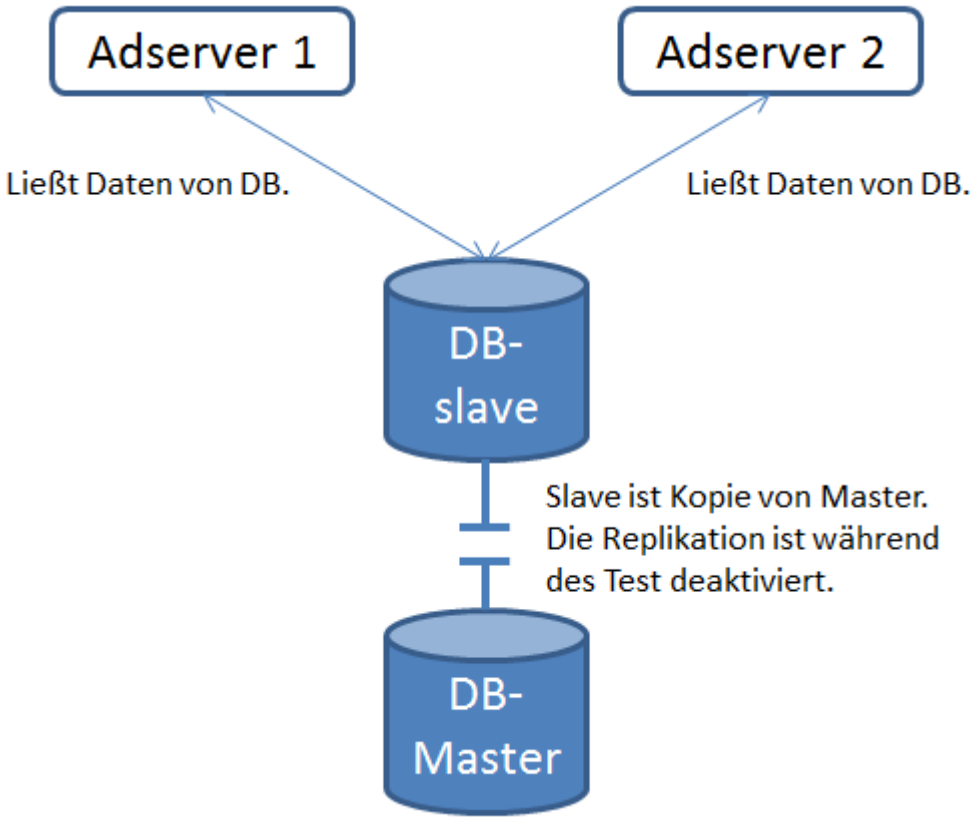

Abbildung 2.3.: Schematische Darstellung des Testaufbaus. Beide Adserver greifen auf die selbe Datenbank zu, die ein Slave der Echt-Datenbank ist. Alle Rechner liegen im selben Netzwerk

Da das Problem möglichst isoliert betrachtet werden soll, um keinerlei Messergebnisse zu verfälschen, ist es nicht möglich, Live-Adserver und Live-Datenbanken direkt zu analysieren. Auf diesen finden, neben den originären Funktionen, viele zusätzliche unterstützende Operationen statt, die im Rahmen des Tests aber nicht berücksichtigt werden sollen. Ansonsten wäre es denkbar, Flaschenhälse zu finden, die nicht

direkt mit dem Adserver-Start zu tun haben, so dass eventuell falsche Schlussfolgerungen gezogen werden. Nichtsdestoweniger müssen alle Komponenten, die am Start selbst beteiligt sind, möglichst Live-getreu nachgebildet werden, dies führt zu folgendem Testaufbau:

Der verwendete Datenbank-Server ist ein Slave der tatsächlichen Live-Datenbank, d. h. er ist ein Replikat derselben, und hat einen identischen Datenbestand. Wegen des Replizierungsprozesses kann es allerdings zu einer minimalen und irrelevanten Abweichung kommen. Er ist nicht in das Produktiv-System eingebunden, womit Anfragen außerhalb des Testszenarios ausgeschlossen sind. Damit eignet er sich für den geplanten Einsatzzweck. Da der stattfindende Replikationsprozess für Verkehr auf dem Netzwerk und der Festplatte des Datenbankservers sorgt, und dieser Verkehr unerwünscht ist, muss der Replikationsprozess zunächst deaktiviert werden. Auf zwei anderen Rechnern im Netzwerk werden eigens konfigurierte Adserver installiert. Als deren Datenquelle wird der Datenbank-Server eingerichtet. Ihre übrige Konfiguration ist so gewählt, das keinerlei tatsächliche Interaktion mit dem Live-System stattfindet. Dies ist für den Test auch unerheblich, da nur der Adserver-Start selbst zu simulieren ist. Somit sind auch keine negativen Effekte zu befürchten, die das Produktiv-System behindern könnten. Werden die Adserver nun gestartet, kann der Test beginnen.

Zunächst wird aber der tatsächliche Leerlauf-Datendurchsatz des Netzwerks berechnet. Zur Messung wird das Konsolen-Programm „netcat“ verwendet. Dieses wird sowohl auf dem Datenbank-Server als auch auf einem der beiden anderen Rechner installiert. Dann wird netcat dazu verwendet, Pakete von einem zum anderen Rechner zu senden, wobei netcat auf einem der Rechner auf die Pakete horcht. Nach rund 10 Sekunden kann der Prozess abgebrochen werden, und aus der Menge gesandter Daten in Bytes und der Dauer der Verbindung lässt sich der Datendurchsatz berechnen. Dieser ergibt 66 Megabyte/s. Zu erwarten ist ein Netto-Datendurchsatz von ca. 117,5 Megabyte/s (Ahlers, 2007) bei komplett isolierten Systemen. Nun sind zwar die beiden Endpunkte der Messung von anderen Zugriffen getrennt, nicht aber das Netzwerk selbst. So ist es durchaus möglich, das der gemessene Wert nicht dem Leerlauf-Datendurchsatz entspricht, und höher ist. Als Grundlage für weitere Berechnungen wird aber weiterhin vom gemessenen Datendurchsatz von 66 Megabyte/s ausgegangen.

Das tatsächliche Profiling wird mit drei weiteren Programmen durchgeführt:

1. Der Datenbank-Server selbst bietet über die einsehbare Prozessliste[3] die Möglichkeit, eingehende Anfragen zu betrachten, und deren Dauer festzuhalten. In der Prozessliste werden alle aktiven Verbindungen aufgeführt, damit assoziierte SQL-Statements, und der Zustand der Verbindung. Zu erwarten sind zwei Verbindungen, zu den beiden Rechnern auf denen die Adserver installiert sind.
2. Ebenfalls eingesetzt wird das Konsolen-Programm dstat[4]. Dstat kann genutzt werden, um die Auslastung aller System-Ressourcen aufzuzeichnen. Insbesondere die prozentuale Beanspruchung aller Prozessorkerne, und die Menge des Netzwerk-Verkehrs in Megabyte ist hier von besonderen Interesse, bieten diese beiden Kennzahlen doch Hinweise auf Engpässe der Hardware oder des Netzwerks. Aufgezeichnet wird im Sekundentakt.
3. Um eine Überlastung des Eingabe/Ausgabe-Systems zu messen, wird das Programm iostat[5] eingesetzt. Über das E/A-System kann Peripherie angesprochen werden, die mit Hilfe von E/A-Bausteinen an das interne System angeschlossen ist, und mit diesem über Eingabe- und Ausgabebefehle kommuniziert. Zugriffe auf die Festplatte werden z. B. über das E/A-System durchgeführt (Hellmann, 2013). Da letztlich alle benötigten Daten auf der Festplatte abgelegt sind, könnte somit ein potentieller Flaschenhals identifiziert werden. Konkret von Belang sind hier die Lese- und Schreibzugriffe, gemessen im Zwei-Sekunden-Takt.

Die Auswertung wird zuerst mit einem Adserver wiederholt, danach wird ein weiterer Adserver zusätzlich gestartet um zu überprüfen, ob gleichzeitige Starts nachteiligen Einfluss auf die Gesamtlaufzeit haben.

2.3.1.2. Gesamtlaufzeit des Adserver-Starts

Sowohl mit einem als auch mit zwei Adservern wurde eine Gesamtdauer von 6:17 Minuten gemessen. Es ist also zumindest bei zwei gleichzeitigen Starts keine negative Beeinflussung festzustellen. Dazu angemerkt sei, dass wie bereits erwähnt, zusätzlicher und vor allen auch schwankender Verkehr im Netzwerk nicht komplett zu vermeiden ist. Insofern ist es durchaus möglich, dass der Start mit zwei Adservern bei konstanter Netzwerkauslastung länger gedauert hätte. Anders ausgedrückt kann es sein, dass zu diesem Zeitpunkt wenig Verkehr im Netzwerk vorlag, was zur gleichen

[3]Siehe auch http://dev.mysql.com/doc/refman/5.1/en/processlist-table.html
[4]Siehe auch http://linux.die.net/man/1/dstat
[5]Siehe auch http://linux.die.net/man/1/iostat

Startdauer führt. Das ist insbesondere deswegen wahrscheinlich, da der Datenbankserver die Anfragen der beiden Adserver seriell abarbeitet, und daher zwangsläufig Wartezeiten anfallen. Dies scheint aber eine so geringe Signifikanz zu haben, das eine nachteilige Beeinflussung zumindest beim gleichzeitigen Start von zwei Adservern nicht messbar ist. Das der Start mehrere Adserver erst bei vielen gleichzeitigen Starts an Einfluss gewinnt, zeigt bereits die historische Auswertung (siehe Kapitel 2.1). Ein echter Live-Adserver mit der gleichen Datenquelle, der nicht von anderen Einflüssen isoliert ist, braucht für denselben Start rund 900 Sekunden, und damit deutlich länger. Hier macht sich der Testaufbau bemerkbar. Da aber die implementierten Verbesserungen mit demselben Aufbau getestet werden, ist Vergleichbarkeit gegeben.

2.3.1.3. Auswertung der Prozessliste des MySQL-Servers

Wie zu erwarten, werden in der Prozessliste zwei Verbindungen angezeigt. Die einzigen enthaltenen SQL-Statements sind die beiden Abfragen, die für das Laden der Banner beider Adserver verantwortlich sind. Eine dieser Anfragen ist in ca. drei Sekunden abgearbeitet, in Summe sind dies sechs Sekunden. Danach versetzen sich beide Verbindungen in den Zustand „sleeping“. Alle anderen SQL-Anfragen werden dermaßen schnell bearbeitet, dass sie erst gar nicht in der Prozessliste sichtbar sind. Es ist zwar zu sehen, dass sich die Zeit für zwei Adserver verdoppelt, da wir aber eine Gesamtdauer des Starts von 6:17 Minuten haben, können einzelne, rechenintensive SQL-Abfragen nicht die Ursache für das Problem sein, da der Datenbank-Server offensichtlich in der Lage ist, die Anfragen sehr schnell zu bearbeiten.

2.3.1.4. Auswertung mit dstat

Bei der Auswertung der CPU-Auslastung mittels dstat ist darauf zu achten, keine Durchschnitts-Auslastung aller CPU-Kerne zu errechnen, sondern Einzelwerte für jeden Kern. Die Ursache ist die, dass beim Starten des Adservers nahezu alle Aufrufe seriell erfolgen, also nacheinander, da eine Parallelisierung des Starts noch nicht vorgenommen worden ist. Entsprechend ist zu erwarten, dass maximal immer nur ein CPU-Kern ausgelastet ist. Ist das der Fall, handelt es sich um einen Hardware-Engpass. Wird stattdessen nun ein Mittelwert über alle acht Kerne gebildet, von denen nur einer am Limit arbeitet, ist es mathematisch nicht mehr möglich,

den Engpass noch wahrzunehmen. Daher ist es sinnvoll, das Maximum der CPU-Auslastung über alle Kerne hinweg zu bestimmen. Zusätzlich zur CPU-Auslastung ist auch die Netzwerk-Auslastung sehr interessant.

Wird das SQL-Statement, dass für das Laden der Banner benötigt wird, vom Datenbank-Server ausgewertet, ist zu beobachten, dass ein CPU-Kern zu ca. 50 % ausgelastet ist. Dies dauert für einen Adserver rund drei Sekunden an, und entspricht damit der Zeit, die zuvor auch in der Prozessliste zu beobachten war. Danach ist ein starker Anstieg des Netzwerk-Verkehrs festzustellen, da die abgefragten Daten nun zum Adserver gesendet werden. Dieser dauert für einen Adserver etwa zehn Sekunden an. In dieser Zeit wird versucht, mehr Daten zu versenden, als Netzwerk-Kapazität zur Verfügung steht. In Summe sollen rund 750 Megabyte an Daten versendet werden, was bei einem Datendurchsatz von 66 Megabyte/s etwas mehr als elf Sekunden dauert. Über die gesamte Dauer von 6:17 Minuten wird ziemlich genau ein Gigabyte an Daten vom Datenbank-Server versendet. Das entspricht einer minimalen Dauer von 15,5 Sekunden, um alle Daten tatsächlich zu versenden. Wird ein weiterer Adserver gleichzeitig gestartet, wird das doppelte an Daten versendet, also zwei Gigabyte. Entsprechend sollte dies mindestens 31 Sekunden dauern. Ein Anstieg der CPU-Auslastung ist bei gleichzeitigen Zugriff von zwei Adservern nicht zu erkennen.

Daraus lassen sich zwei Erkenntnisse ableiten:

1. Es ist kein CPU-Limit vorhanden, da keine der Prozessorkerne auch nur ansatzweise ausgelastet war.
2. Jeder Adserver benötigt minimal 15,5 Sekunden, um alle benötigten Daten über das Netzwerk zu erhalten. Es entsteht hier also ein handfester Engpass. Bei einem gleichzeitigen Start von fünf Adservern dauert alleine das Laden der Daten schon deutlich über eine Minute. Im Kontrast dazu steht aber die Gesamt-Startdauer des Adservers von 6:17 Minuten. Gemessen daran, macht das Laden der Daten also nur einen kleinen Teil aus, und kann nicht das Hauptproblem darstellen. Insofern ist auch die Netzwerk-Übertragungsrate nicht das eigentliche Problem.

2.3.1.5. Auswertung mit iostat

Über iostat lässt sich nachweisen, ob zu viele E/A-Operationen zum Problem werden. Dazu listet das Programm Informationen wie Schreib- und Lesezugriffe pro Sekunde

auf. Die Anzahl der möglichen Zugriffe auf eine Festplatte wird maßgeblich durch die Dauer, die für die Umpositionierung der Leseköpfe der Festplatten benötigt wird, beeinflusst. Der Datenbankserver verfügt über drei Festplatten gleicher Kapazität im RAID 5 Verbund. Das heißt die Daten werden über alle Festplatten verteilt und es wird ein Paritätsblock je Datenblock geschrieben. Das erhöht die Sicherheit, da in diesem Falle eine Festplatte ausfallen kann, ohne das Daten verloren gehen (Schmidt, 2013). Die Analyse mit iostat hat ergeben, das tatsächlich im Schnitt unter 20 dieser Zugriffe je Sekunde erfolgen. Es sind auch keine relevanten Maxima während der Messperiode aufgetreten. Als Vergleichswert soll die Anzahl der E/A-Zugriffe auf die Festplatten während einer Belastungsphase dienen. Dazu wird die Replikationen wieder aktiviert. Jetzt wird die Slave-Datenbank mit der Master-Datenbank synchronisiert, so dass ständig neue Daten auf die Festplatten geschrieben werden. Führt man nun dieselbe Analyse durch, ist festzustellen, dass rund 150 Zugriffe pro Sekunde getätigt werden. Das Maximum bei dieser Messung liegt außerdem bei rund 220 E/A-Operationen pro Sekunde. Wenn während des Adserver-Starts nur rund 20 Zugriffe/s erfolgen, in anderen Szenarien aber 150 Zugriffe/s möglich sind, ist die logische Schlussfolgerung, dass das Eingabe/Ausgabe-System nicht überlastet wird. Damit ist auch hier kein Engpass vorhanden.

2.3.1.6. Fazit der Server-seitigen Analyse

Abgesehen von einer kurzfristigen Überlastung des Netzwerks ist kein Problem erkennbar, das darauf hindeutet, dass der Datenbank-Server das Hauptproblem verursacht. Zu erwähnen ist allerdings, dass abseits einer isolierten Testumgebung, in der Praxis viel zusätzlicher Netzwerkverkehr auf einem Datenbank-Server zu erwarten ist. Das lastet die Datenbank weiter aus, so dass weniger Spielraum zur Verteilung der übrigen Last vorhanden ist.

2.3.2. Client-seitige Analyse

Da die server-seitige Analyse keine signifikanten Engpässe aufweist, ist es sinnvoll, den Client, in diesem Fall der Adserver, genauer hinsichtlich der Verarbeitungsdauer zu untersuchen. Es gibt einige sinnvolle Werkzeuge, um Schwachstellen im Programmcode zu identifizieren. Diese ermöglichen es, Methoden zu identifizieren, die zu einer starken CPU-Belastung führen, oder auch Speicherlecks zu finden. Da

für die Auswertung im ersten Schritt nur von Bedeutung ist, wie lange der Adserver für welchen Arbeitsschritt benötigt, sollte auf derartige spezifische Analysen jedoch verzichtet werden. Stattdessen wird der Code manuell um Timer ergänzt, die die Dauer jedes Arbeitsprozesses messen. Das Ergebnis wird nun wiederum in ein Log geschrieben. Die Hauptarbeitsschritte, wie Kapitel 2.1 zu entnehmen ist, werden bereits gemessen. Da die Auswertung dieser Messung ergeben hat, das rund 85 % des gesamten Startvorgangs auf das Laden der Banner entfällt, reicht es aus, nur diese näher zu betrachten. Jetzt ist die Frage zu stellen, wie dieser Arbeitsschritt weiter zu unterteilen ist, um eine detaillierte Aussage treffen zu können. Da mittels der clientseitigen Analyse die Geschwindigkeit der SQL-Abfrage und des Adserver-Codes evaluiert werden sollte, ergeben sich daraus zwei zentrale Punkte:

1. Dauer der SQL-Abfragen zum Laden der Objekte und der Unterobjekte aus der Datenbank. Dieser Schritt umfasst implizit die Verarbeitung im MySQL-Server und den Netzwerk-Verkehr.
2. Dauer der Verarbeitung der Objekte durch den Adserver-Code, also das eigentliche Erstellen der Objekte.

Daher wird jede SQL-Abfrage mit einem Timer versehen, ebenso wie jede Objekterstellung, die aus dieser Abfrage resultiert. Eine weitere Granulierung ist zumindest vorerst nicht sinnvoll. Mit dieser Modifikation werden fünf Live-Adserver unterschiedlicher Farmen versehen und gestartet. Anschließend ist es möglich, deren geschriebene Logs zu analysieren, indem die Zeiten kumuliert und in eine prozentuale Relation gebracht werden.

2.3.2.1. Ergebnisse der Auswertung

Wird die Bannerlade-Dauer hinsichtlich der Erstellung von Objekten und der Datenbankzugriffe analysiert, so ist festzustellen, dass je nach Adserver ca. 65-70 % der Gesamtdauer auf die Datenbankzugriffe entfällt. Weitere 15 % entfallen auf das Erstellen der Objekte selbst. Die restliche Zeit wird für unterstützende Aufgaben aufgewendet, die nicht explizit zugeordnet werden konnte. Da die beiden letztgenannten Punkte verhältnismäßig wenig Anteil an der Gesamtdauer haben, ist klar, dass die Datenbankabfragen, also deren Netzwerk-Kommunikation und die Verarbeitung in der Datenbank selbst, das Hauptproblem darstellen. Wichtig ist anzumerken, dass dies Durchschnittszahlen sind. Da es unterschiedliche Bannertypen gibt, für die unterschiedlich komplexe Verarbeitungsschritte im Adserver-Code durchgeführt

werden, kann je nach tatsächlichem Datenbestand, die Dauer der Objekterstellung steigen und damit der Anteil der Datenbankzugriffe an der Gesamtdauer zurück gehen. Damit nimmt dann auch die Objekterstellung eine signifikantere Stellung ein.

2.4. Fazit

Wie sind die Ergebnisse zu interpretieren? Die server-seitige Analyse ergab, dass außer einer relevanten aber nicht kritischen Netzwerk-Überlastung kein Engpass auf dem Datenbank-Server bzw. dem Netzwerk entsteht. Die client-seitige Analyse hingegen legt nahe, dass die meiste Zeit für Datenbank-Abfragen und die zwischengelagerte Netzwerk-Kommunikation aufgewendet wird. Diese beiden auf den ersten Blick gegensätzlichen Ergebnisse ergeben bei näherer Betrachtung der SQL-Befehle durchaus Sinn.

Nachdem das Hauptobjekt aus der Datenbank geladen worden ist, in dieser Betrachtung z. B. ein Banner, werden alle Unterobjekte dieses Banners über einzelne SQL-Statements ebenfalls aus der Datenbank nachgeladen. Dies wird für alle Objekte wiederholt. Da bei der client-seitigen Analyse auch alle SQL-Abfragen einzeln gemessen worden sind, ist es über die Gesamtanzahl der Messungen auch möglich, eine Aussage über die gesamte Menge der abgesetzten SQL-Statements zu treffen. Je nach Adserver und Farm differiert der Wert zwar stark, aber gerade für die kritischen Farmen liegt die Anzahl bei mehreren hunderttausend einzelnen Anfragen. Zwar wird für diese Anfragen der Verbindungskontext zum MySQL-Server immer gehalten, er muss also nicht ständig neu aufgebaut werden, das Problem ist aber folgendes:

Jede einzelne Abfrage unterliegt Netzwerk-Latenzen, die sich über deren Gesamtanzahl kumuliert. Messbar ist diese Latenz z. B. mit dem Diagnose-Werkzeug „ping“. Über „ping“ lässt sich nicht nur feststellen, ob ein Server und ein Client (oder auch zwei Hosts ganz allgemein) über ein Netzwerk verbunden sind, und über dieses miteinander kommunizieren können. Zusätzlich wird auch noch die so genannte „Round trip time“, kurz auch RTT genannt, angegeben. Die Round trip time beschreibt die Dauer, die zwischen dem Senden eines Netzwerk-Pakets und dem Erhalten der passenden Antwort durch den Kommunikationspartner liegt. Deren Wert beträgt in diesem Netzwerk 0,15 Millisekunden. Für den realistischen Wert von 400.000 ein-

zelnen Anfragen ergibt das in Summe bereits eine Minute, und das schon, wenn davon ausgegangen wird, dass je Abfrage die Latenz zur einmal anfällt, was jedoch nicht zu erwarten ist. Daher ist bereits hier von wenigen Minuten auszugehen. Hinzu kommt dann noch die Verarbeitung, also das Durchführen der eigentlichen Abfrage, durch den MySQL-Server. Zwar werden die einzelnen SQL-Statements so schnell verarbeitet, dass diese in der Prozessliste (siehe Kapitel 2.3.1.3) erst gar nicht auftauchen, bei mehreren hunderttausend Anfragen summieren sich aber bereits wenige Millisekunden zu Minuten auf. Damit lässt sich dann auch erklären, warum augenscheinlich kein Engpass auf dem Datenbank-Server vorliegt, die Kommunikation mit diesem aber trotzdem zum Hauptproblem wird. Um dies zu verifizieren, wurde ein Programm geschrieben, welches die Zeit misst, die für die Ausführung von hunderttausend SQL-Anfragen benötigt wird. Damit ist es möglich, dass Problem reduziert auf dessen Kern zu betrachten. Das Ergebnis beläuft sich auf rund 30 Sekunden. Wird mit einbezogen, dass eine sehr simple, aber realistische Abfrage zur Überprüfung verwendet wurde, keinerlei anschließende Verarbeitung stattfindet, und zusätzlich einige hunderttausend mehr Abfragen beim Starten des Adservers getätigt werden, dann lässt sich die Annahme bestätigen.

Was bedeutet das für die angesprochenen möglichen Ursachen? Genannt wurden:

1. Überlastetes Netzwerk
2. Überlastete Hardware
3. Langsame SQL-Abfragen
4. Langsamer Adserver-Code

Zusammenfassend lässt sich sagen, dass das Problem eine Kombination aus einem überlasteten Netzwerk und nicht optimierten SQL-Abfragen ist, die wiederum zu mehr Netzwerk-Verkehr führen, als nötig wäre, ist. Für diese Probleme gilt es eine Lösung zu finden. In bestimmten Fällen kann auch die Objekterstellung selbst zeitaufwändig sein (siehe Abschnitt 2.3.2.1). Dieser Punkt wird aber fürs Erste ausgeklammert, da er im Vergleich weniger ausgeprägt ist. Sollte die erste durchgeführte Optimierung jedoch erfolgreich sein, so ist zu vermuten, dass in diesem Falle die Objekterstellung selbst die meiste Zeit aus macht, da alle übrigen Engpässe dann deutlich weniger Anteil an der Rest-Startdauer haben. Daher ist es sinnvoll diese Annahme später zu überprüfen um gegebenenfalls einen detaillierteren Blick auf die Objekterstellung zu werfen.

3. Lösungsalternativen

Nachdem das Problem hinreichend analysiert wurde, können unterschiedliche Lösungsalternativen aufgelistet und bewertet werden. Auf Basis dieser Bewertung soll anschließend eine Methode ausgewählt werden, die es umzusetzen gilt. Es wurde festgestellt, dass der Flaschenhals eine Kombination aus unterschiedlichen Problemen ist, daher kann auch das Lösungsverfahren eine Kombination mehrerer Ansätze sein. Zunächst sollte aber die erfolgversprechendste Variante ausgewählt werden, um sich auf diese zu konzentrieren. Ergänzende Maßnahmen sollten dennoch als solche kenntlich gemacht werden, um diese gegebenenfalls nachträglich hinzuzufügen.

Bevor die einzelnen Verfahren evaluiert werden, sollte ein Blick darauf geworfen werden, wieso es überhaupt zu Neustarts kommt. Dies hat Auswirkungen auf die Effizienz unterschiedlicher Verfahren. Im Allgemeinen sind drei unterschiedliche Szenarios denkbar:

1. Geplantes Update: Im Rahmen des Produktlebenszyklus ist eine neue Version mit zusätzlichen Features und Produktverbesserungen entwickelt worden. Diese hat ihren finalen Release-Zustand erreicht, und soll nun für alle Kunden zur Verfügung stehen. Daher wird eine neue Version des Adservers eingespielt, und dieser neu gestartet. Derartige Neustarts sind vorhersehbar und damit planbar. Es ist daher mögliche, Updates zu festgelegten Zeiten auszuführen, zu denen der langsame Neustart eines Adservers kein Problem darstellt, z. B. des Nachts, wo verhältnismäßig wenig Adrequests eingehen. Daher stellen geplante Updates auch kein Problem dar.
2. Absturz: Ein kritischer Programmfehler führt zu einem Absturz einzelner oder mehrerer Adserver. Es ist nicht möglich Daten des Adservers zu sichern. Der Neustart ist unerwartet erforderlich und daher nicht planbar. Die Analyse der Logs hat ergeben, dass Abstürze je Adserver maximal dreimal pro Monat auftreten, und das die Wahrscheinlichkeit des Auftretens derartiger Abstürze

tendenziell sinkt. Drei Abstürze pro Monat sind als selten zu bewerten.

3. Neustart, bei Erkennung eines Problems: Möglich wäre z. B., dass keine Werbekampagnen mehr ausgeliefert werden. Tritt ein derartiges Problem auf, kann der Adserver regulär herunter gefahren werden, und die Geschäftsobjekte können gefahrlos gesichert werden, so lange sie nicht selbst Teil des Problems sind. Es lassen sich zwei Subtypen erkennen, deren Unterschied hinsichtlich des Aufwands der Lösung des aufgetretenen Problems zwar signifikant ist, die aber im technischen Kontext dieser Arbeit als Einheit betrachtet werden können, eben weil in beiden Fällen der Adserver normal heruntergefahren wird.

   a) Ein Problem wird durch einen Mitarbeiter oder Kunden erkannt. Zwar muss der Fehler zeitnah behoben werden, der Zeitpunkt des Neustarts sowie die Anzahl der neu zu startenden Adserver kann aber flexibel bestimmt werden. Derartige Probleme sind daher zwar unerwartet, aber eingeschränkt planbar. So können Downtimes von Adserver minimiert werden, und das Problem als solches wird entschärft.

   b) Der Neustart wird durch ein externes Programm initiiert, welches einen Fehler erkannt hat. In diesem Fall geschieht der Neustart automatisch, und es ist nicht möglich, einzugreifen.

3.1. Vermeiden von Neustarts

Das zentrale Problem ist die Dauer des Neustarts eines Adservers, insofern wäre der sinnigste Ansatz, den Neustart als solches komplett zu vermeiden, oder diesen zumindest so zu legen, dass die lange Startdauer kein Problem darstellt. Letzteres ist für geplante Updates sinnvoll, wird aber im Rahmen des Update-Managements bereits so gehandhabt, und kann daher ignoriert werden. Soll der komplette Neustart entfallen, kommt als erstes die Verwendung von Dynamic Libraries in Frage. In Linux-basierten Systemen wird hier auch von einer Shared Library gesprochen, auch wenn die Unterscheidung der beiden Begriffe zunehmend an Bedeutung verliert. Je nach Dynamic Linking Mode können Shared Libraries entweder zur Load-Time geladen werden - dann stehen sie schon mit Start des Prozesses zur Verfügung - oder zur Runtime (Laufzeit). Insbesondere letzteres bietet den Vorteil, dass Bibliotheken bei Bedarf während der Laufzeit einer Anwendung geladen und auch geändert werden können. Wird eine vom Adserver benötigte Programmbibliothek aktualisiert, kann

diese so neu eingespielt werden, und steht somit dem Adserver zur Verfügung. Dies erfordert allerdings manuelle Aufrufe der Shared Libraries über API-Funktionen im Code, statt diese, wie bisher, automatisiert zu laden (Stevanovic, 2014). Dies erhöht daher nicht nur im beträchtlichen Maße die Komplexität des Codes, sondern geht auch zu Lasten der Performance. Des Weiteren bietet diese Methode keine Vorteile, falls ein Adserver abstürzt. Aus den genannten Gründen ist daher die Vermeidung eines Neustarts über diesen Ansatz nicht sinnvoll möglich, daher sollten Alternativen in Betracht gezogen werden.

3.2. Einsatz verteilter Datenbanken

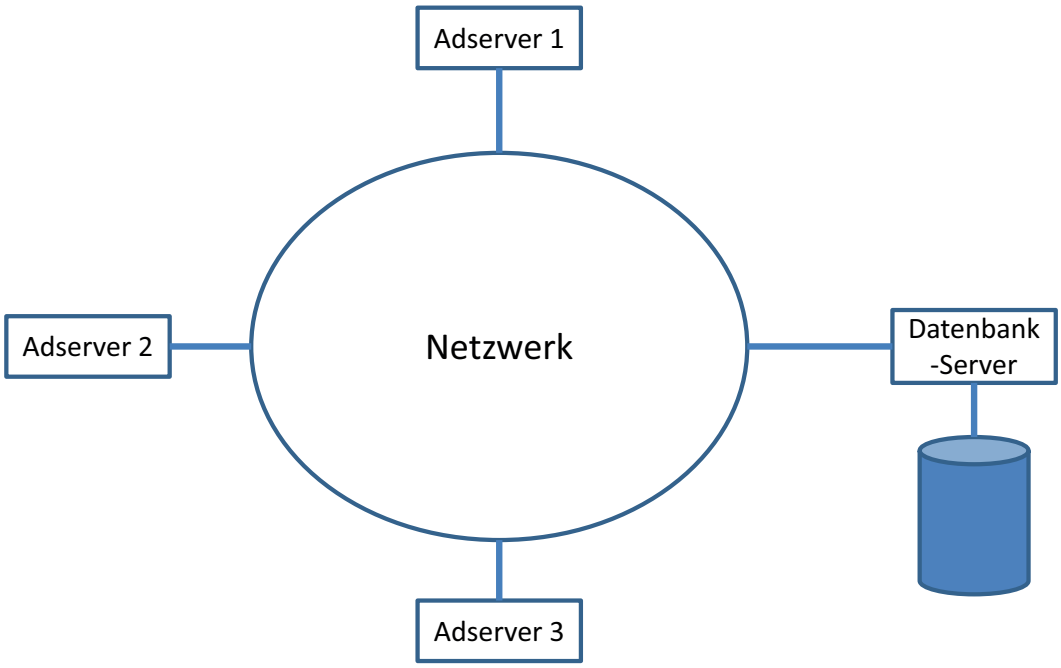

Abbildung 3.1.: Struktur eines zentralen Datenbanksystems

Abbildung 3.1 stellt schematisch die aktuelle Struktur der Kommunikationswege zwischen Adservern und der Datenbank während des Neustarts dar. Alle Adserver kommunizieren über dasselbe Netzwerk mit einer zentralen Datenbank. Wie festgestellt wurde, kommt es zu Netzwerkengpässen aufgrund der begrenzten Transferrate des Netzwerks, und zu anfallenden Latenzen aufgrund hunderttausender übermittelter SQL-Anfragen. Eine Alternative wird in Abbildung 3.2 dargestellt. Diese zeigt

einen beispielhaften Aufbau der Kommunikation, wenn statt einer zentralen Datenbank eine verteilte Datenbank zum Einsatz kommt. Eine verteilte Datenbank ist eine Sammlung von mehreren, logisch zusammenhängenden Datenbanken, die über ein Netzwerk verteilt sind (Özsu, 2007). Statt wie bisher alle Adserver-Instanzen auf dieselbe Datenquelle zugreifen zu lassen, liegt auf jedem Rechner auf dem ein Adserver befindlich ist, eine Datenbank, die die notwendigen Daten für diesen Adserver bereithält. Dies kann über eine horizontale Fragmentation der Gesamtmenge der Daten erfolgen, indem nur eine Teilmenge der Tupel einer Relation am jeweiligen Knoten gespeichert wird, oder über eine vertikale Fragmentation, bei der eine Teilmenge der Attribute der ursprünglichen Relation gespeichert wird (Özsu, 2007). Es wäre im Sinne der Definition zwar nicht notwendig, tatsächlich jeden Adserver mit einer eigenen Datenbank auszustatten, aufgrund der geografischen Nähe dieses Ansatzes, wäre die Geschwindigkeitssteigerung aber am höchsten. Startet ein Adserver, so muss er zu diesem Zeitpunkt nur die Daten abrufen, die sich bereits auf dem Rechner befinden, bzw. die im Knoten der verteilten Datenbank gespeichert sind. Die Kommunikation über das Netzwerk entfällt vollkommen und der Start sollte signifikant beschleunigt werden, da Datenzugriffe auf die Festplatte bedeutend schneller erfolgen. Weil die Datenbanken allerdings logisch zusammenhängen, und viele Adserver mit dem gleichen Datenbestand starten, ist bei Schreibvorgängen eine anschließende Synchronisation aller bzw. der meisten Datenbanken erforderlich. Das Netz selbst wird dadurch ebenfalls belastet. Dies spielt für den eigentlichen Start aber keine Rolle, da ja nur lesend auf die vorrätigen Daten zugegriffen wird. Insofern stellt dies kein Problem dar.

Eine nähere Betrachtung der Hauptvorteile verteilter Datenbanken kann Aufschluss darüber geben, ob diese überhaupt für die Lösung der angesprochenen Problematik in Frage kommen. M. T. Öszu nennt in seinem Buch „Principles of Distributed Database Systems“ vier fundamentale Vorteile verteilter Datenbanken (Özsu, 2007):

1. Transparente Verwaltung verteilter und replizierter Daten.
2. Verlässlicher Datenzugriff über verteilte Transaktionen.
3. Verbesserte Performance.
4. Einfache Systemerweiterung.

Ein weiterer wichtiger Punkt ist der Aufwand, der benötigt wird, um eine unter Echt-Bedingungen lauffähige Lösung basierend auf verteilten Datenbanken zu realisieren.

Im nachfolgenden sollen diese fünf Punkte betrachtet, und deren Relevanz für die Problemstellung determiniert werden.

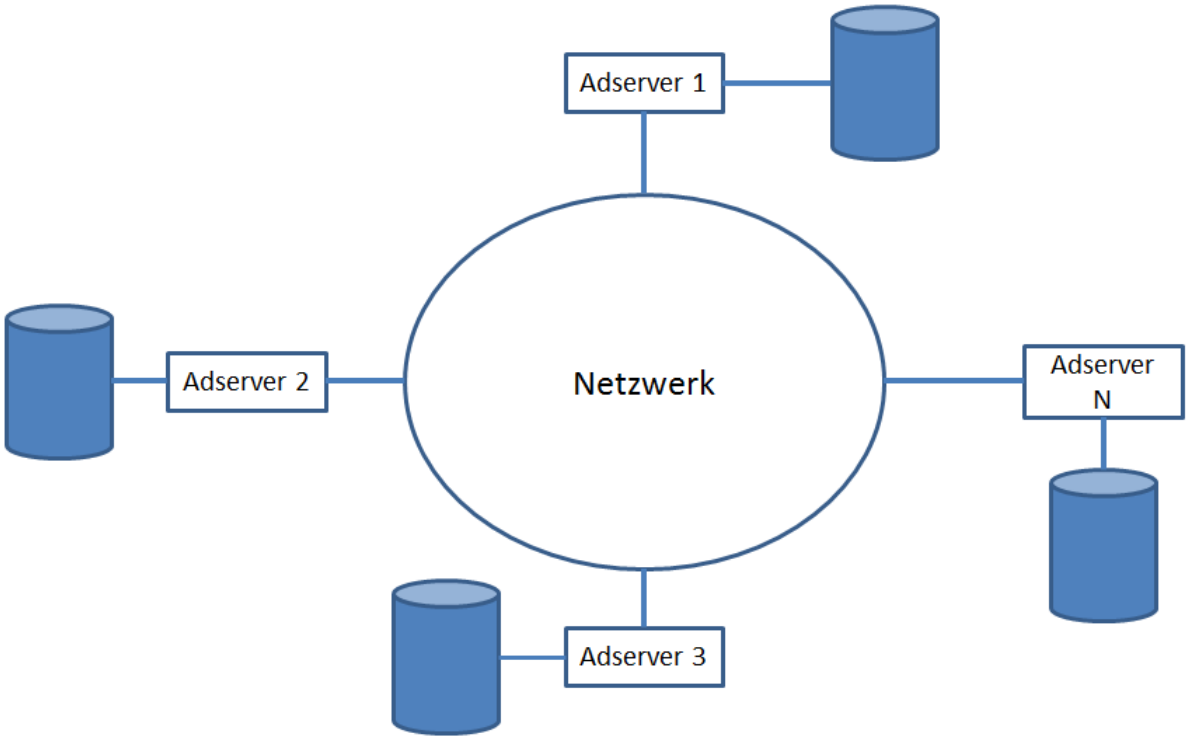

Abbildung 3.2.: Stuktur eines verteilten Datenbanksystems. Die Rechner Adserver 1 bis N können auch als Knoten bezeichnet werden, an denen eine Teilmenge der Daten gespeichert ist.

3.2.1. Transparente Verwaltung der Daten

Transparenz meint im Kontext verteilter Datenbanken nichts anderes, als das Implementierungsdetails vor dem Benutzer „versteckt“ werden (Özsu, 2007). Als Beispiel sei folgendes SQL-Statement gegeben, dass die Gehälter aller Mitarbeiter einer Firma um zehn Prozent anheben soll:

UPDATE mitarbeiter_gehalt SET gehalt = gehalt * 1.1;

Für eine zentrale Datenbank ist der Sachverhalt klar und einfach, da sich alle betreffenden Daten in derselben physischen Datenbank befinden. Werden diese Daten aber nun horizontal fragmentiert, z. B. wenn das Unternehmen mehrere Standorte weltweit hat, und jeder dieser Standorte seine eigenen Mitarbeitergehälter verwaltet, dann müssen mehrere physikalisch getrennte, aber logisch verknüpfte Datenbanken

aktualisiert werden. Die transparente Verwaltung der Daten sorgt nun dafür, dass genau das gleiche Statement ausgeführt werden kann. Das verteilte Datenbankmanagmentsystem kümmert sich selbstständig um die Aktualisierung der korrekten Daten, ohne das der Benutzer die interne Verteilung kennen muss. Transparenz einer verteilten Datenbank ist auf verschiedenen Abstraktionsebenen nötig, allen gemein ist, dass der Benutzer keine Kenntnisse über die genaue Implementierung besitzen muss.

Im Kontext des Neustarts eines Adservers besitzt dieser Punkt durchaus Relevanz, da er den Aufwand, der zur Implementierung benötigt wird, verringert. In der Entwicklungs-Phase muss kein Augenmerk darauf gelegt werden, die Daten korrekt zu verteilen, bzw. verteilte Daten anzusprechen, da dies bereits intern durch die Datenbankmanagement-Software erledigt wird. Der Punkt „Aufwand“ wird in Abschnitt 3.2.5 näher beleuchtet.

3.2.2. Verlässlicher Datenzugriff über verteilte Transaktionen

Eine Transaktion überführt eine Datenbank von einem konsistenten Zustand in einen anderen, ebenfalls konsistenten, Zustand. Als Beispiel dient wieder die Aktualisierung der Mitarbeitergehälter. Fatal wäre das Auftreten eines Fehlers während des Aktualisierens der Gehälter, so dass nur die Hälfte der Mitarbeiter ein höheres Gehalt bekommt. Daher wird ein derartiges Statement in einer Transaktion gekapselt. Diese sorgt dafür, dass entweder alle Datensätze erfolgreich geändert werden, oder gar keine. Die Daten sind daher immer konsistent. Transaktionen betreffen nur Schreibvorgänge auf der Datenbank. Beim Neustart eines Adservers wird nun zwar ausschließlich lesend auf die Daten zugegriffen wird, so isoliert sollte der Punkt aber nicht betrachtet werden. Alle Daten die gelesen werden, wurden zuvor auch geschrieben, und spätestens dann ist verteilte Transaktionssicherheit von Nöten. Setzt man daher auf verteilte Datenbank, kann das nicht ausgeklammert werden. Dies spricht also für verteilte Datenbanken, da die Konsistenz der Daten immer garantiert ist.

3.2.3. Verbesserte Performance

Kernprinzip einer verteilten Datenbank ist die horizontale Skalierung[1] (Limoncelli u. a., 2014). Durch das Hinzufügen mehrerer Datenbank-Knoten, können SQL-Anfragen von diesen Knoten parallel bearbeitet werden, so dass die Resultate schneller zur Verfügung stehen. Zusätzlich können Programme auf den Datenbank-Knoten zugreifen, der ihnen physikalisch am nächsten ist, so dass Latenzen minimiert werden, und ebenfalls eine Geschwindigkeitssteigerung erzielt wird. Im Idealfall, zumindest aus Performance-Sicht, befindet sich ein Datenbank-Knoten auf demselben Rechner wie der Adserver selbst, so dass keine Netzwerk-Kommunikation während des Starts von Nöten ist. Es ist in diesem Fall aber zu beachten, dass die Synchronisation nach wie vor über das Netzwerk erfolgt, und die Datenbank auf der Festplatte fortwährend aktualisiert wird, was ebenfalls zu Lasten der Geschwindigkeit gehen kann.

Um vorwegzugreifen: Es gibt verteilte Datenbanken, wie z. B. Aerospike[2], die intern dieselben Mechanismen zur Geschwindigkeitssteigerung verwenden, wie die, die letztlich im Rahmen dieser Arbeit implementiert worden sind (siehe Abschnitt 3.3.2). Dazu kommt die Möglichkeit, horizontale Skalierung zur Geschwindigkeitssteigerung einzusetzen. Unter dieser Prämisse ist der Einsatz einer verteilten Datenbank aus Performance-Sicht durchaus attraktiv.

3.2.4. Einfache Systemerweiterung

Verteilte Datenbanken lassen sich durch das Hinzufügen zusätzlicher Knoten relativ einfach erweitern. Dies ist bei Expansion ein wichtiger Faktor, und macht den Einsatz einer solchen natürlich sehr attraktiv. Dennoch gibt es Lösungen, die noch einfacher erweiterbar sind. Dies wird ebenfalls in Kapitel 3.3.2 beschrieben. Daher ist dies kein Argument für verteilte Datenbanken.

3.2.5. Aufwand für Implementierung

Hauptnachteil verteilter Datenbanken, ist der Fakt, dass es sehr aufwändig, ist eine lauffähige Implementierung in der zur Verfügung stehenden Zeit zu erreichen. Es

[1] Nicht zu verwechseln mit der horizontalen Fragmentierung
[2] vgl. http://www.aerospike.com/docs/operations/manage/aerospike/fast_restart.html

gibt sehr viele unterschiedliche verteilte Datenbanken, alleine alle hinsichtlich ihrer Tauglichkeit zu überprüfen, dürfte eine gewisse Zeit in Anspruch nehmen[3]. Da das Ziel aber eine funktionierende Implementierung ist, sprengen verteilte Datenbanken den zeitlichen Rahmen. Dazu kommt, dass diese teilweise kostenpflichtig sind, was sie im Vergleich zu kostenlosen Lösungsansätzen zusätzlich unattraktiver macht.

3.2.6. Fazit zum Einsatz verteilter Datenbanken

Aufgrund dessen, dass die genannten Vorteile weniger schwer wiegen als die Nachteile, spricht nichts für den Einsatz einer verteilten Datenbank, so dass eine einfachere Lösung vorzuziehen ist.

3.3. Lokales Caching der Objekte

Da einer der Hauptfaktoren für die langsame Startzeit die Netzwerk-Komponente ist, wäre es sinnvoll diese komplett zu umgehen. Dazu können die notwendigen Daten lokal auf dem Rechner gespeichert werden. Zwar müssen die Daten immer noch über das Netzwerk mit der Hauptdatenbank synchronisiert werden, aber dies muss nicht zwangsläufig während des Starts erfolgen, so dass dessen Laufzeit verkürzt wird. Als Speicherkomponenten kommen Festplatten oder Arbeitsspeicher in Frage. Beide verfügen über genug Speicherkapazität um alle Daten zu speichern. Die Gesamtgröße aller Geschäftsobjekte wurde bereits in vorigen Kapiteln gemessen, und mit ca. einem Gigabyte beziffert. Des Weiteren sind bei beiden die Transferraten deutlich höher, als dies im Netzwerk der Fall ist. Dies ist Tabelle 3.1 zu entnehmen. Mittlerweile verwenden viele Adserver eine Solid State Disk anstatt einer herkömmlichen HDD, so das nochmals deutliche höhere Transfer-Raten möglich sind.

Beide Varianten, lokales Caching im Arbeitsspeicher oder der Festplatte, haben Nachteile und Vorteile, die nachfolgend miteinander verglichen werden sollen, um die bessere Variante auswählen zu können.

[3]Laut professioneller Meinung diverser Kollegen

Gigabit-Netzwerk (Netto)	Solid State Disk	HDD	DDR3-1600 (PC3-12800)
117,5 MB/s (Ahlers, 2007)	510 MB/s (Shimpi, 2014)	160 MB/s (CDRinfo.pl, 2014)	12,8 GB/s (Kingston Technology, 2014)

Tabelle 3.1.: Transfer-Rate unterschiedlicher Komponenten

3.3.1. Serialisierung in Datei

Für das Caching auf der Festplatte ist eine Serialisierung in eine Datei erforderlich, da die eigentlichen Objekte nicht direkt gespeichert werden können. Grundsätzlich sollte so vorgegangen werden:

1. Es wird ein separater Dienst programmiert, der in bestimmten Zeitabständen die gleichen SQL-Statements ausführt, die bisher vom Adserver selbst ausgeführt werden. Die so erhaltenen Daten werden in eine Datei serialisiert. Da keine kontinuierliche Synchronisation der Daten mit der Datenbank vorgesehen ist, stellen diese daher einen Snapshot der echten Daten dar, und sind somit nicht aktuell. Das Problem das bei kontinuierlicher Synchronisation auftritt, ist die schiere Menge an Adservern, für die der aktuelle Datenbestand zwischengespeichert werden soll. Auch hier würde das Netzwerk wieder als Engpass auftreten. Daher ist es sinnvoller, die Daten in zu definierenden Zeitabständen zu aktualisieren, aber nicht permanent zu synchronisieren.
2. Auf diese kann der Adserver nun zugreifen, um die Objekte zu deserialisieren und zu erstellen.
3. Da die Daten nicht aktuell sind, sorgt nun der Adserver selbst dafür, die Differenz zum aktuellen Objektstand zu laden.

Der kritische Punkt ist das Nachladen der Differenz. Die zentrale Frage lautet: Wie lange dauert dieser Schritt im Vergleich zur ursprünglichen Startdauer eines Adservers? Um dies beantworten zu können, sollte ein Blick auf die Kommunikationswege bei einer Aktualisierung oder Neuerstellung eines Geschäftsobjekts geworfen werden. Dies wird in Abbildung 3.3 veranschaulicht. Wird ein Objekt im Frontend vom Kunden aktualisiert, so wird dies dem XML-Daemon mitgeteilt. Dieser aktualisiert nun die Datenbank, und sendet eine XML-Nachricht an den Adserver, dass sich ein Objekt geändert hat. Zwischen Adserver und XML-Daemon befindet sich eine Message-Queue, so dass keine Nachrichten verloren gehen, selbst wenn der Adserver

mit den bisherigen Objektaktualisierungen ausgelastet ist, und die nachfolgenden noch nicht abarbeiten kann. Mit der Information, welches Objekt neu angelegt oder geändert wurde, kann der Adserver eine entsprechende Anfrage an den SQL-Server schicken, und so seinen Objektstand aktualisieren. Ist die Message-Queue voll, dann werden keine neuen Änderungen mehr berücksichtigt.

Soll gemessen werden, wie lange der Adserver mit Objektaktualisierung beschäftigt ist, kann ein Timer gestartet werden, wenn die Nachricht eingeht, und wieder gestoppt werden, wenn die Aktualisierung abgeschlossen ist. Diese Information wird im Log ausgegeben. Über die Summe aller Aktualisierungen kann ein Mittelwert je Stunde gebildet werden. Dieser schwankt je nach Adserver zwischen 180 und 600 Sekunden. Im Gesamtkontext der Fragestellung ist daher folgende Aussage zu treffen: Für jede Stunde, die die serialisierte Datei nicht aktualisiert wurde, dauert das Nachladen der Differenz zusätzliche 180 bis 600 Sekunden. Vergleicht man dies zur Gesamtstartdauer von 400 bis 900 Sekunden, wird schnell klar, dass dies substantiell zu lange dauert. Diese Methode ist so daher ohne weiteres nicht sinnvoll integrierbar.

Diesem Nachteil stehen jedoch auch Vorteile gegenüber:

1. Auch nach einem Neustart des Rechners stehen die Daten noch zur Verfügung.
2. Da ein separater Dienst die Daten aus der Datenbank lädt, ist es unerheblich, wenn der Adserver selbst abstürzt. Die Daten bleiben weiterhin verfügbar.

Die Vorteile sind durchaus nicht gering, gerade im Vergleich zur Verwendung des Arbeitsspeichers zur Speicherung der Objekte. Damit diese Methode aber überhaupt sinnvolle Geschwindigkeitsvorteile bringt, müssten die Kommunikationswege bei der Objektaktualisierung überarbeitet bzw. optimiert werden. Dies ist vorerst jedoch nicht vorgesehen, daher scheidet diese Variante aus. Im Kapitel „Ausblick“ soll darauf eingegangen werden, welche zusätzlichen Methoden zur Verfügung stehen, dieses Vorgehen praktikabel zu integrieren.

3.3.2. Serialisierung in den Arbeitspeicher

Ein Blick auf Tabelle 3.1 lässt klar werden, dass der Arbeitsspeicher mit Abstand den höchsten Datendurchsatz zulässt. Insofern ist die zu erwartende Geschwindigkeitssteigerung, sollte dieser zur Speicherung der Objekte verwendet werden, am höchsten.

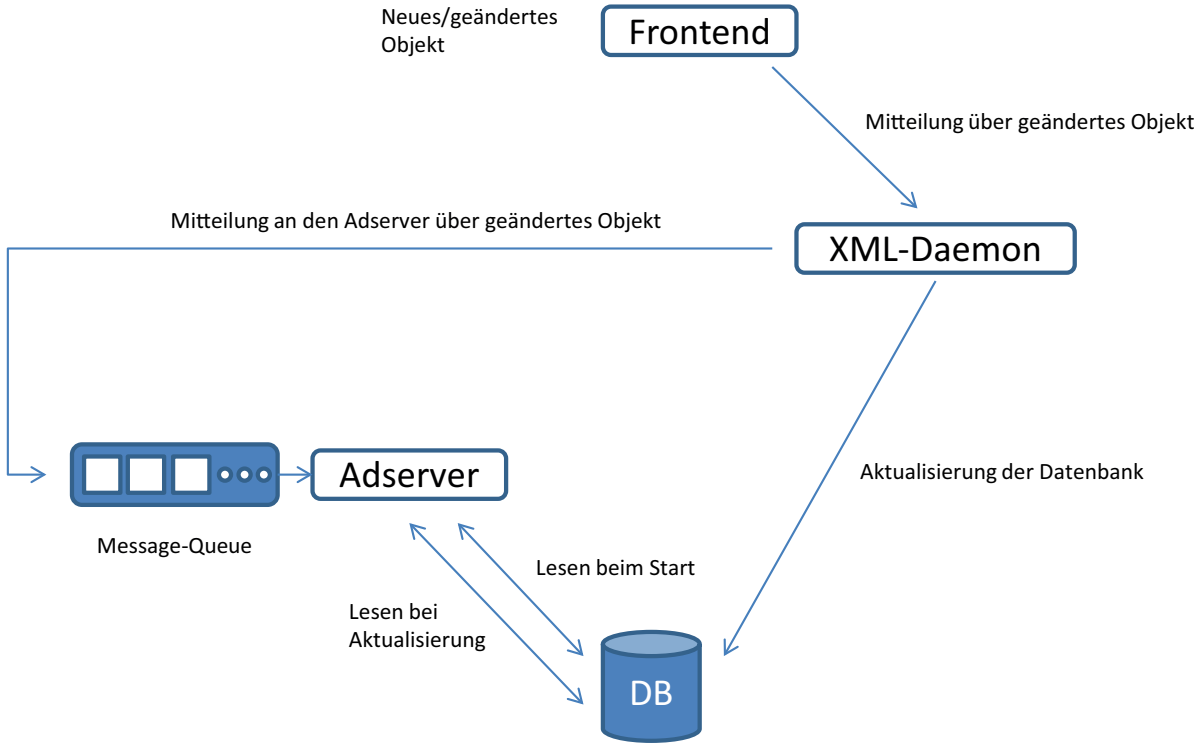

Abbildung 3.3.: Vereinfachte Darstellung einer Objektaktualisierung

Nun werden die Geschäftsobjekte zur Laufzeit des Adservers sowieso schon im Hauptspeicher gehalten. Das Problem ist die Lebenszeit des verwendeten Speicherbereichs. So lange der Adserver-Prozess aktiv ist, steht diesem sein eigener Speicherbereich zur Verfügung (Gray, 2003). In diesem liegen dann auch alle Objekte. Wird der Adserver aber nun Heruntergefahren, steht der Speicherbereich wieder zur freien Verfügung für alle übrigen Prozesse. Anders gesagt: Die Objekte sind weg. Es gilt also einen Weg zu finden, die Daten im Arbeitsspeicher auch über die Prozess-Lebensdauer hin zu persistieren. Hierfür kann Shared Memory verwendet werden. Shared Memory erlaubt es mehreren Prozessen, einen gemeinsamen virtuellen Speicherbereich zu verwenden, und ist damit ein Mittel zur Interprozess-Kommunikation. Verwenden mehrere Prozesse ein so genanntes Shared Memory Segment, dann ist der letzte Prozess, der diesen Bereich verwendet, für gewöhnlich dafür zuständig diesen wieder zu löschen (Gray, 2003). Das ist für meist auch der Prozess, der das Segment erstellt hat. Nun gibt es im gegebenen Szenario zwar nur einen Prozess, der wichtige Punkt ist aber der, dass der Speicherbereich aktiv gelöscht werden muss. Auch wenn der Adserver-Prozess beendet wird, bleibt der Speicherbereich mit allen enthaltenen Objekten weiterhin reserviert. Shared Memory hat zumindest in einer Linux-Umgebung

Kernel-Persistenz, und verschwindet damit erst, wenn der Rechner heruntergefahren wird. Damit ist es ein geeignetes Mittel, um die Objekte zu persistieren. Das Vorgehen beim Verwenden dieser Technik wäre wie folgt:

1. Der Adserver lädt beim ersten Start alle Objekte wie bisher von der Datenbank.
2. Wird der Prozess regulär heruntergefahren (siehe Kapitel 3), dann werden alle aktiven Objekte in ein Shared Memory Segment serialisiert. Auch hier ist eine Serialisierung erforderlich. Die Gründe werden im nächsten Kapitel näher beschrieben.
3. Startet der Adserver nun wieder, kann er die Objekte aus dem Shared Memory laden. Das Nachladen einer Differenz ist nicht erforderlich, da die Dauer für einen Neustart so kurz ist, dass alle Differenzen automatisch über die Message-Queue bezogen werden können (siehe Abbildung 3.3), da diese noch alle Änderungen vorhält.

Der Hauptnachteil ist offensichtlich. Stürzt der Adserver während der Laufzeit ab, werden die Objekte nicht korrekt ins Shared Memory geschrieben, und das erneute Laden von der Datenbank ist erforderlich. Wie in der Einführung zu Kapitel 3 aber bereits beschrieben, tritt dieses Problem selten auf. Daher verspricht diese Methode den höchsten Geschwindigkeitszuwachs, bei moderaten Nachteilen. Außerdem ist die Erweiterbarkeit einfacher als bei verteilten Datenbanken, da der Adserver selbst optimiert ist, und keinerlei externen Komponenten benötigt werden. Es reicht aus, einen neuen Adserver zu installieren und es gibt nichts Weiteres zu beachten.

3.4. Restrukturierung der SQL-Statements

Wie bereits in Kapitel 2 festgestellt, werden einige hunderttausend einzelner SQL-Anfragen an den Datenbank-Server geschickt, wodurch ständig neu auftretende Latenzen in der Kommunikation zum Problem werden. Besser ist es, einige wenige SQL-Anfragen zu senden, und deren Rückgabe dann im Speicher zu halten. Das könnte zu einem nennenswerten Geschwindigkeitsschub führen. Da allerdings die Implementierung des Shared Memory-Ansatzes den Netzwerk-Verkehr komplett umgeht, und die Optimierung der SQL-Statements somit überflüssig macht, ist die Restrukturierung

vorerst wenig sinnvoll. Im Hinterkopf ist allerdings zu behalten, dass keine Daten im Shared Memory gespeichert werden, wenn ein Adserver abstürzt. Dann muss dennoch von der Datenbank geladen werden, und die Optimierung der SQL-Statements spielt wieder eine Rolle. Als zusätzliche Verbesserung ist es also durchaus angebracht, auch diesen Punkt zu implementieren.

3.5. Fazit und Auswahl der Methode

Unter Betrachtung aller Aspekte, ergibt es am meisten Sinn, alle Objekte ins Shared Memory zu serialisieren. Die Serialisierung in eine lokale Datei ist durchaus sinnig, zuerst müsste jedoch die Art wie dem Adserver Objektveränderungen mitgeteilt werden verändert werden, da dieses Vorgehen ansonsten zu langsam wäre. Ebenfalls zweckmäßig wäre eine Überarbeitung der SQL-Statements. Dies wird vorerst jedoch nicht angestrebt, da die Verwendung des Shared Memory einen größeren Performance-Gewinn verspricht.

Eine Betrachtung technischer Lösungsansätze, wie z. B. der Ausbau des Netzwerks zu einer schnelleren Variante, wurde ausgelassen, weil die Expertise fehlt, innerhalb des zeitlichen Rahmens, fundierte Alternativen anzubieten.

4. Konzeption

Der Vergleich unterschiedlicher Lösungsalternativen hat ergeben, dass das lokale Caching der Geschäftsobjekte im Arbeitsspeicher den höchsten Geschwindigkeitszuwachs bringen dürfte. Um die Objekte zu persistieren wird Shared Memory eingesetzt. Bevor mit der Implementierung der Funktionalität begonnen werden kann, muss zunächst ein Konzept erarbeitet werden, das die Besonderheiten des gewählten Ansatzes berücksichtigt. Außerdem ist es sinnvoll, einen Prototyp zu entwickeln, um die Korrektheit des Konzepts zu verifizieren und vorab Testwerte für die Geschwindigkeit zu ermitteln. Dieser Prototyp sollte die elementarsten Bausteine enthalten, die später auch implementiert werden sollen.

4.1. Besonderheiten Shared Memory

Geplant ist es, die Geschäftsobjekte beim Herunterfahren des Adservers vom prozessinternen Speicherbereich in ein bzw. mehrere Shared Memory Segmente zu verschieben. Die so angelegten Segmente werden dann beim Neustart wieder ausgelesen, und wiederum in den prozessinternen Speicherbereich zurück geholt. Auf den ersten Blick scheint das ziemlich umständlich zu sein. Es wäre doch viel sinnvoller, die Geschäftsobjekte direkt im Shared Memory anzulegen, und mit diesen dann zur Laufzeit des Adservers zu arbeiten. Sowohl der Kopiervorgang beim Herunterfahren, als auch der Kopiervorgang beim Neustarten entfällt dann vollkommen, und es kann Zeit gespart werden. Es gibt allerdings einige Einschränkungen bei der Verwendung von Shared Memory, die diesen Zwischenschritt unabdingbar machen. Die Dokumentation zur Bibliothek boost::interprocess verdeutlicht die zentralen Einschränkungen (Gaztanaga, 2012):

1. Es werden keine Zeiger auf Objekte unterstützt: Ein Zeiger enthält nicht das Objekt selber, sondern die Speicheradresse des Objektes, womit dieses dann

wieder referenziert werden kann. Nun besteht das Problem, das Shared Memory Segmente, werden sie mit einem laufenden Prozess verknüpft, für jeden Prozess eine unterschiedliche Speicheradresse haben, auch dann, wenn es das gleiche Segment ist. Auch der Zugriff auf ein Shared Memory Segment erfolgt dabei über einen Zeiger. Ein Beispiel: Prozess A verknüpft das Shared Memory Segment mit dem Schlüssel 1000 (ein Identifikator für das Segment) in seinen eigenen Speicherbereich. Die Adresse dieser Verknüpfung ist #BF0006. Nun verknüpft Prozess B dasselbe Segment wiederum in seinen Speicherbereich. Nun ist die Adresse allerdings #BC007C. Aufgrund der anderen Adresse würden Zeiger auf absolute Speicherbereiche nicht funktionieren. Was allerdings gleich bleibt, ist die Verschiebung der Objekte, relativ zur Startadresse. Hier spricht man auch von Offset. Befindet sich ein Objekt, z. B. ein Werbebanner, 60 Byte hinter dem Beginn des Shared Memory Segments, ist die unterschiedliche Startadresse egal, da die Verschiebung um 60 Byte konstant bleibt. Es wäre hier noch möglich, statt mit absoluter Adressierung mit relativer Adressierung zu arbeiten. Dazu können Offset-Zeiger verwendet werden (die zumindest durch die Bibliothek boost::interprocess auch standardmäßig angeboten werden).

2. Es werden keine Referenzen zu Objekten unterstützt: Ist das Objekt mit den Namen „auto1“ eine Referenz auf das Objekt „auto2“, dann ist auto1 ein Alias für auto2, es ist das gleiche Objekt. Da diese intern ebenfalls als Zeiger implementiert werden, kommt es zu den gleichen Problemen.

3. Virtualität wird nicht unterstützt: Virtuelle Tabellen und Zeiger zu virtuellen Tabellen sind die Grundlage für Polymorphie und Vererbung. Auch hier ein Beispiel: Die Klasse „Kreis“ erbt Methoden von der Klasse „Form“. In C++ wird den Methoden der Klasse Form das Stichwort „virtual“ vorangestellt, um zu indizieren, das diese Methoden in einer Kindklasse überschrieben werden können. Neben den eigenen Klassenvariablen und Methoden haben nun beide Klassen zusätzlich einen Zeiger auf eine virtuelle Tabelle, in der alle Methoden aufgelistet sind, die diese tatsächlich verwendet. Hat die Klasse „Form“ eine Methode „zeichne()“, die von „Kreis“ mit einer eigenen Implementierung überschrieben wird, dann enthält die virtuelle Tabelle der Klasse „Kreis“ auch einen Zeiger zur eigenen Implementierung, und nicht zu der aus der Elternklasse (Stroustrup, 2014). Zeiger zu virtuellen Tabellen sind aber nicht direkt sichtbar, und werden immer im Speicherbereich des eigenen Prozesses abgelegt.

Das führt zu folgendem Szenario, würden die tatsächlichen Objekte im Shared Memory abgelegt werden: Während die Objekte im Shared Memory liegen, befinden sich die zugehörigen virtuellen Tabellen immer noch im Speicherbereich des eigentlichen Prozesses. Wird dieser beendet, gehen auch die virtuellen Tabellen verloren. Startet der Adserver jetzt neu, verfügen alle Objekte, die von Vererbung Gebrauch machen, über einen Zeiger zu einer virtuellen Tabelle, die nicht mehr existiert. Das Programm stürzt ab.

Da die drei nicht unterstützen Funktionalitäten in C++ sehr gebräuchlich sind, und auch für die Geschäftsobjekte des Adservers intensiv gebraucht werden, kann daher das eigentliche Objekt nicht im Shared Memory abgelegt werden. Es muss eine Alternative Variante gefunden werden, bei der die Objekte serialisiert ins Shared Memory verschoben werden. Die Serialisierungs-Methode wird in Abschnitt 4.1.1 näher erläutert. Die angesprochene boost::interprocess Bibliothek stellt einen Wrapper um systemeigene Funktionen zum Anlegen und Schreiben von Shared Memory Segmenten dar. Diese wird zur Zeit nicht vom Adserver unterstützt. Die angesprochenen Einschränkungen gelten aber genau so.

Ein weiteres Problem ergibt sich durch die Anforderungen an die Größe der Shared Memory Segmente. Es gelten hier zwei Einschränkungen, die bei der Konzeption berücksichtigt werden müssen:

1. Die Größe eines Shared Memory Segments beträgt immer das Vielfache der Page-Size, oder auch Speicherblock-Größe (Gaztanaga, 2012). Diese beträgt auf den verwendeten Rechnern 4 KB. Daraus folgt, das nicht für jedes einzelne Objekt ein eigenes Shared Memory Segment angelegt werden sollte. Ist ein Objekt 100 Byte groß, und ein Segment wird nur für dieses Objekt angelegt, dann werden 3996 Byte (4 * 1024 Byte - 100 Byte) verschwendet. Bei ein paar 100.00 Objekten wäre die Speicherplatzvergeudung enorm. Es müssen also Blöcke von Objekten in gemeinsame Segmente geschrieben werden.

2. Es gibt eine maximale Größe für Shared Memory Segmente. Diese beträgt 32 MB auf den Testrechnern. Diese lässt sich zwar theoretisch erhöhen, aber es ist besser mehrere kleinere Segmente zu haben, als ein großes. Grund dafür ist folgender: Es wird rund ein Gigabyte an Speicherplatz benötigt (siehe Abschnitt 2.3.1.4), der nicht auf einmal reserviert werden sollte, damit es nicht zu Engpässen kommt.

Punkt 2 führt auch zu einer weiteren Konsequenz: Wenn Speicher gespart werden soll, dürfen die Objekte nicht kopiert werden, bzw. nicht alle zugleich. Das würde folglich bedeuten das zwei Gigabyte Arbeitsspeicher reserviert sind, einmal für die Objekte die noch im Speicherbereich des Adservers liegen, und für die serialisierten Kopien im Shared Memory. Um das zu vermeiden werden die Objekte serialisiert, ins Shared Memory kopiert, und dann wird das Original direkt gelöscht. Dass das für jedes Objekt einzeln geschieht, wird nur für dieses kurzfristig doppelter Speicherplatz belegt, was kein Problem ist. Dieses Vorgehen spielt für ein anderes Problem eine Rolle. Es existieren drei unterschiedliche Objekttypen, die relevant sind. Jede davon besitzt Referenzen auf andere Objekte:

1. Filter: Hier gibt es unterschiedliche Typen (siehe auch Abschnitt 2.1 zur Beschreibung von Filtern), von denen alle ohne nennenswerten Aufwand serialisierbar sind. Zu beachten ist lediglich der inverse Filter. Er invertiert die gewünschte Bedingung anderer Filter. Nehmen wir einen Filtertypen an, der dafür sorgt das nur Seitenbesucher, die den Browser Firefox verwendet, die Werbung angezeigt bekommen. Wird dieser Filter in einen inversen Filter geschachtelt, kehrt sich die Bedingung um. Alle Seitenbesucher außer Firefox-Nutzer können die Werbung sehen. Daraus resultiert, dass der inverse Filter Referenzen zu anderen Filtern besitzt. Diese werden beim Starten des Adservers geprüft, und wenn der referenzierte Filter nicht gefunden wird, wird er aus der Datenbank nachgeladen. Letzteres soll vermieden werden. Daher müssen beim Starten alle Filter geladen werden, außer inverse Filter. Deren serialisierte Daten werden stattdessen in Listen gemerkt, um sie dann zu erstellen, wenn bereits alle übrigen Filter geladen worden sind. So existiert die Referenz bereits zuvor, und die Kommunikation mit der Datenbank wird umgangen.
2. Banner: Diese besitzen Referenzen zu Filtern.
3. Werbeplätze: Werbeplätze wiederum referenzieren sowohl Filter als auch Banner.

Wird ein Objekt wie geplant gelöscht, nachdem es serialisiert und ins Shared Memory geschrieben wurde, kann es zu Problemen kommen. Man nehme an das alle Filter in Shared Memory Segmente verschoben wurden, und damit gelöscht sind. Nun soll das gleiche für Banner wiederholt werden. Deren Referenzen zu Filtern müssen ebenfalls gesichert werden, um sie später wieder herstellen zu können. Nun existieren die referenzierten Filter aber nicht mehr. Die Information ist verloren. Daher ist

die Reihenfolge sehr wichtig. Beim Herunterfahren müssen die Geschäftsobjekte in der Reihenfolge Werbeplätze, Banner und Filter verarbeitet werden. Beim Starten hingegen genau umgekehrt, also Filter, Banner und dann Werbeplätze. So sind alle Referenzen verfügbar.

Eine letzte Herausforderung ist die Ermittlung der Größe der Blöcke von Objekten. Die maximale Größe der Segmente beträgt 32 MB und soll ausgereizt, aber nie überschritten werden, sonst kommt es zu Abstürzen. Auch soll ein Shared Memory Segment erst dann erstellt werden, wenn es vollständig gefüllt werden kann. Die Größe eines serialisierten Objektes ist aber erst bekannt, wenn die Serialisierung tatsächlich abgeschlossen ist. Daher werden die Objekte serialisiert, und zunächst in eine Liste geschrieben, wobei fortwährend die Summe der Größe aller Objekte gemerkt werden muss. Kommt ein Objekt zur Liste hinzu, so dass 32 MB überschritten werden, kann ein Segment angelegt werden und die Objekte werden hineingeschrieben. Alle bis auf das letzte Objekt, da dieses nicht mehr hineinpasst. Die Liste wird exklusive des letzten Elements gelöscht, und wird dann wieder befüllt, bis der nächste Block fertig ist, oder aber keine weiteren Daten mehr vorhanden sind.

4.1.1. Serialisierung mittels Apache Thrift

Objekte müssen serialisiert werden, da sie nicht direkt in Shared Memory Segmenten abgelegt werden können. Bisher erfolgt die Serialisierung von bestimmten Log-Daten des Adservers mittels Apache Thrift, deswegen ist es sinnvoll, dieses Framework ebenfalls zu verwenden. Das Know-How ist bereits vorhanden, und Thrift gilt als sehr schnell[1], wenn es um Serialisierung geht. Apache Thrift ist ein Framework für skalierbare cross-language Dienste, und ermöglicht die Kommunikation von Anwendungen, die in unterschiedlichen Programmiersprachen implementiert wurden (Apache Software Foundation, 2014). Um das zu ermöglichen generiert Thrift automatisiert Schnittstellen zu unterschiedlichen Sprachen (C++, Java, Python...), die über eine sogenannte Thrift-Definition beschrieben werden. Das folgende Beispiel wird auch für den Prototypen beschrieben in Abschnitt 4.2 weiter verwendet.

Szenario: Ein Objekt vom Typ Person soll von einer Anwendung so serialisiert werden, das eine weitere Anwendung dieses Format problemlos lesen kann. Die Klas-

[1] siehe Abschnitt 4.2

se Person verfügt nur über zwei Klassenvariablen: Vorname und Nachname. Eine Thrift-Definition für dieses Beispiel sieht wie folgt aus:

```
namespace cpp person

struct PersonThrift
{
    1: required string vorname,
    2: required string nachname,
}
```

Die Klasse wird „PersonThrift“ genannt, ihr zugeordnet werden die beiden Variablen „vorname“ und „nachname“, wobei es sich bei beiden um einen string handelt. Der Zusatz „required“ sagt aus, dass jedes serialisierte Objekt vom Typ „PersonThrift“ immer über Vorname und Nachname verfügt. Alternativ kann der Zusatz „optional“ verwendet werden. In diesem Fall müssen die Daten nicht zwangsläufig vorhanden sein. Thrift erlaubt einige grundlegenden Datentypen, die in allen unterstützen Programmiersprachen vorkommen. Das sind aber keineswegs alle. Die bereits angesprochenen Zeiger und Referenzen werden auch von Thrift nicht unterstützt, da nicht alle Programmiersprachen dieses Prinzip kennen. Das spielt später noch eine wichtige Rolle. Ist die Thrift-Definition erstellt, kann über einen Konsolenbefehl entsprechender Code erstellt werden, der für die Ziel-Sprache read- und write-Methoden enthält. Das Beispiel geht davon aus, dass die Definition in einer Datei namens „person.thrift“ liegt.

```
thrift -r --gen cpp person.thrift
```

Während der erste Teil „thrift -r –gen“ nur indiziert, das Thrift Code erstellen soll, und der letzte Teil „person.thrift“ der Name der Date ist, steht „cpp“ dafür, dass die Zielsprache in diesem Fall C++ ist. Anschließend steht eine Klasse zur Verfügung, die es für die Sprache C++ ermöglicht, die Klasse „person“ zu serialisieren und zu deserialisieren[2]. Die Serialisierungs-Methode gibt außerdem die Größe des serialisierten Objektes zurück, was für die Bestimmung der Segmentgröße (siehe 4.1) wichtig ist.

[2]Die Header-Datei der so erstellten Klasse ist in Anhang A.1 zu sehen

4.2. Prototyp

Bevor mit der eigentlichen Implementierung begonnen wird, ergibt es Sinn, einen Prototyp zu erstellen, der die zentralen Funktionen hinsichtlich ihrer Umsetzung überprüft Des Weiteren ist es so möglich, bereits vorab zu analysieren, wie hoch der Zeitaufwand für Serialisierung und Deserialisierung mittels Apache Thrift ist. Im Prototypen werden nur die zwei Hauptpunkte berücksichtigt:

1. Das Serialisieren eines existierenden Objekts mittels Apache Thrift.
2. Das Schreiben des serialisierten Objekts in ein Shared Memory Segment.

Grundlage des Beispiels ist die Thrift-Definition aus 4.1.1. Der grundlegende Vorgang ist wie folgt in den Kommentaren des Codes (C++) beschrieben, und wird anschließend erklärt:

```
// System-Bibliothek für Shared Memory Verwaltung
#include <sys/shm.h>

// Klassen und Bibliotheken für Thrift
#include "person_types.h"
#include "person_constants.h"
#include <thrift/protocol/TBinaryProtocol.h>
#include <thrift/transport/TBufferTransports.h>

// Klasse für Person
#include "Person.h"

// sonstige Includes
#include <stdio.h>
#include <string>

using namespace apache::thrift;
using namespace boost;

int main(int argc, char** argv)
{
    // Objekt im Speicherbereich des eigenen
```

```
    //   Prozesses anlegen
    // Objekt in Thrift-generierte Klasse kopieren
    // Objekt serialisieren
    // Kopieren der Daten ins Shared Memory
    // Segment von Prozess lösen und löschen
}
```

Einzig erwähnenswerter Include ist die Header-Datei „shm.h“. Diese bietet alle Funktionen, die für die Verwaltung von Shared Memory Segmenten notwendig sind. Zunächst wird das ein Objekt im Speicherbereich des eigenen Prozesses angelegt:

```
// Objekt im Speicherbereich des eigenen
//   Prozesses anlegen
Person* pers1 = new Person("Dieter", "Schmidt");
```

Die zugehörige Klasse „Person“ ist sehr simpel, und verfügt nur über die beiden Klassenvariablen „vorname“ und „nachname“, daher wird auf nähere Erklärung verzichtet. Anschließend wird dieses Objekt in die automatisch erstellte Thrift-Klasse kopiert:

```
// Objekt in Thrift-generierte Klasse kopieren
PersonThrift personThrift;
personThrift.nachName = pers1->getNachname();
personThrift.vorName = pers1->getVorname();
```

Die Klasse PersonThrift verfügt über die automatisch generierten read und write-Methoden, so dass die Daten jetzt serialisiert werden können:

```
// Objekt serialisieren
shared_ptr<TMemoryBuffer> transport(new TMemoryBuffer);
shared_ptr<TBinaryProtocol> protocol(
    new TBinaryProtocol(transport)
);
personThrift.write(protocol.get());
uint8_t* pbuf;
uint32_t size;
transport.get()->getBuffer(&pbuf, &size);
transport->flush();
transport->close();
```

Die beiden ersten Zeilen weisen Thrift dazu an, die Daten in einen Speicher-Puffer zu schreiben (TMemoryBuffer), und zwar formatiert als Binärdaten (TBinaryProtocol). Mit diesen Informationen kann die write Methode ausgeführt werden, und die Daten sind serialisiert. Die Methode transport.get()->getBuffer schreibt die so generierten Daten in die Variable „pbuf“, und deren Größe in die Variable „size“. Beide Daten sollen nun ins Shared Memory geschrieben werden:

```
// Kopieren der Daten ins Shared Memory
char* shared_memory_start_pointer;

// Shared Memory Segment anlegen über shmget,
// shared_memory_id als Identifikator
if ((int shared_memory_id =
     shmget(1000, size + sizeof(uint32_t), IPC_CREAT | 0666))
        != -1)
{
    // Segment verknüpfen über shmat
    shared_memory_start_pointer = static_cast<char*> (
        shmat(shared_memory_id, 0, 0)
    );
    if (shared_memory_start_pointer == (char*) - 1)
    {
        return false;
    }

    // Position festlegen um Daten zu schreiben
    char* sizeRegion = shared_memory_start_pointer;
    char* dataRegion = shared_memory_start_pointer
        + sizeof(uint32_t);

    // Kopieren der Daten ins Shared Memory
    std::memcpy(sizeRegion, &size, sizeof (uint32_t));
    std::memcpy(dataRegion, &pbuf, size);
}
```

Um ein Shared Memory Segment anzulegen, wird die Methode „shmget“ aus der vorher inkludierten Header-Datei „shm.h“ verwendet. Sie erwartet als Eingabepara-

meter einen Schlüssel, sozusagen der Name des Segments, dessen Größe, und Informationen ob das Segment erstellt oder geöffnet werden soll, sowie die Zugriffsrechte auf das Segment. Die Größe setzt sich zusammen aus der Größe des Objekts (size), und der Menge an Bytes, die zur Speicherung der Größe benötigt werden. shmget liefert eine ID zurück, die nachfolgend das Segment identifiziert. Mit dieser Identifikationsnummer ist es möglich, das Segment über die Methode „shmat“ in den eigenen Speicherbereich zu verknüpfen. Zurückgeliefert wird ein void-Zeiger (also ein typloser Zeiger), der in einen char-Zeiger konvertiert wird. Chars haben eine Größe von einem Byte, was es vereinfacht, Zeigerarithmetik zu benutzen. Diese wird verwendet um die Regionen zu berechnen, in denen die Größe des Objekts und dessen Daten geschrieben werden soll. Die Größe wird an den Anfang des Segments geschrieben (sizeRegion), die Daten selbst vier Byte dahinter (dataRegion), da vier Byte die Menge an Speicher ist, die benötigt wird um zuvor die Größe zu speichern. Über die Funktion „memcpy“ werden nun die Daten aus dem Speicherbereich des eigenen Prozesses in das Shared Memory Segment verschoben. Schließlich muss der über „shmat“ eingebunden Speicherbereich wieder freigegeben werden (shmdt), womit das Segment dann wieder gelöscht werden kann (shmctl):

```
shmdt(shared_memory_start_pointer);
shmctl(shared_memory_id, IPC_RMID, 0);
```

Ausgelesen werden können die Daten analog, in dem die umgekehrte Reihenfolge verwendet wird (siehe auch Anhang A.2). Ein erweiterter Prototyp wurde verwendet, um mit realitätsnahen Testdaten zu ermitteln, wie lange die Serialisierung und Deserialisierung benötigt. Dabei wurde festgestellt, dass sie gemessen an der Dauer für die Objekterstellung, extrem niedrig ist. Das bestätigt, dass Thrift die richtige Wahl ist, und keinen neuen Engpass verursachen sollte. Es wurden nur wenige Geschäftsobjekte zum Testen verwendet, da aber die Gesamtanzahl dieser Objekte in einem Live-Adserver bekannt ist, kann abgeschätzt werden, wie lange der komplette Startvorgang unter Verwendung des Shared Memories dauert. Dabei wurde ein Wert von 90 bis 120 Sekunden extrapoliert. Nur 8,3 Sekunden entfallen dabei auf die Serialisierung bzw. Deserialisierung.

Shared Memory Segmente lassen sich über den Kommandozeilen-Befehl „ipcs“ anzeigen (siehe Abbildung 4.1). Die Spalte „nattch“ indiziert hier, dass keinerlei Prozesse aktuell auf das Shared Memory Segment zugreifen, da der Zähler immer 0 ist.

```
(run-env)mbehnke@mbehnkes:~$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00002775 327688     adserver   666        400        0
0x00004e86 360457     adserver   666        33553915   0
0x00004e87 393226     adserver   666        33529700   0
0x00004e88 425995     adserver   666        33545949   0
0x00004e89 458764     adserver   666        33545665   0
0x00004e8a 491533     adserver   666        33549867   0
0x00004e8b 524302     adserver   666        33542085   0
0x00004e8c 557071     adserver   666        31178370   0
0x0000759d 589840     adserver   666        521487     0
0x00009ca4 622609     adserver   666        86         0
```

Abbildung 4.1.: Liste der Shared Memory Segmente, erstellt von einem Adserver.

4.2.1. Vermeidung von Objektkopien

Ein Problem fällt auf, wenn der Prototyp betrachtet wird. Das eigentliche Objekt „Person“ wird zunächst in die Thrift-Klasse „PersonThrift“ kopiert, bevor es serialisiert werden kann. Dieser Schritt scheint auf den ersten Blick überflüssig, und sorgt für einen höheren Zeitaufwand. Stattdessen wäre es sinnvoller, dass eine Klassenvariable der Klasse „Person“ selbst vom Typ „PersonThrift“ ist (siehe Abbildung 4.2). Dann muss der Konstruktor der Klasse „Person“ zwar so geändert werden, dass er „PersonThrift“ gleich mit initialisiert, um zu serialisieren genügt es dann aber, personThrift.write() aufzurufen. Der Kopiervorgang entfällt völlig. Alle Operationen, auf den Vornamen oder den Nachnamen, die bisher direkt auf die Klasse „Person“ angewendet wurden, werden nun intern so umgeleitet, dass stattdessen Vorname und Nachname der Klassenvariable „personThrift“ manipuliert werden. Wenngleich dieses Vorgehen wesentlich besser anmutet, sorgt die Limitierung von Thrift dafür, dass das Kopieren nicht ohne zusätzlichen Aufwand umgangen werden kann. Wie in Abschnitt 4.1.1 beschrieben, unterstützt Thrift nicht alle Datentypen, die von C++ verwendet werden. Es ist daher nicht möglich, die aktuelle Klassendefinition durch eine Thrift-Klasse zu ersetzen. So verfügen z. B. Banner über eine Klassenvariable vom Typ vector<Filter*>. Dieser enthält Zeiger zu allen zugeordneten Filtern. Nun kann zwar der Vektor in Thrift abgebildet werden, auch wenn anderen Namen für den Typen verwendet werden, der Zeiger selbst ist aber nicht abbildbar. Eine Alternative wäre es, den Vektor so umzuändern, dass er stattdessen den Typen vector<int> hat. Statt einem Zeiger wird dann die Identifikationsnummer des Filters abgespeichert. Dann muss aber zur Laufzeit nach dem Objekt gesucht werden, dass

diese ID hat. Zusätzlicher Zeitaufwand entsteht.

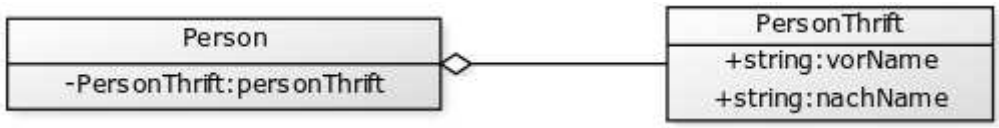

Abbildung 4.2.: Klasse Person mit PersonThrift als Member

Das ist aber nicht das größte Problem. Thrift unterstützt keine Vererbungshierarchien. Betrachten wir das in Abbildung 4.3 dargestellte Beispiel. „HtmlBanner“ verfügt neben seinen beiden Klassenvariablen auch noch zusätzlich über die Klassenvariable von „Banner“. Bei diesem Konstrukt besteht keine einfache Möglichkeit mehr, die entsprechenden Variablen durch eine Thrift-Klasse zu verwenden. Sowohl „HtmlBanner“ als auch „Banner“ bräuchten dann ihre zugehörige Thrift-Klasse als Klassenvariablen, dann ist es aber nicht mehr möglich durch einen einzigen Aufruf der Serialisierungsmethode write() ein zusammenhängendes Objekt zu serialisieren. Es entstehen zwei separate Strings[3]. Diese müssen so im Shared Memory abgespeichert werden, dass die Zusammengehörigkeit erkennbar ist. Dann ist es möglich beide Strings so zu deserialisieren, dass wieder ein zusammenhängendes Objekt entsteht. Auch wenn beide Probleme umgangen werden können, ist der Aufwand der dazu betrieben werden müsste relativ hoch, und potentiell besteht die Gefahr, an anderer Stelle Zeit zu verschwenden. Daher ist geplant, weiterhin das Objekt in die Thrift-Klasse zu kopieren, um diese anschließend zu serialisieren. Ist die Implementierung abgeschlossen, ist es erforderlich zu messen, wie viel Zeit das Kopieren in Anspruch nimmt. Erst wenn dies zu lange dauert, ergibt es Sinn, über die hier angesprochenen Probleme und deren Lösung nachzudenken.

Abbildung 4.3.: Schematisch dargestellte Vererbung zwischen Bannern.

[3]Das serialisierte Objekt wird als String dargestellt, bzw. als Zeichenfolge.

4.3. Klassendiagramm

Aus den definierten Anforderungen und den Erfahrungen die bei der Entwicklung des Prototyps gemacht wurden, kann nun ein Klassendiagramm modelliert werden. Das in Abbildung 4.4 dargestellte Diagramm soll hinsichtlich der Funktion der Klassen beschrieben werden, da die Beschreibung jeder einzelnen Methode den Rahmen sprengen würde. Die Interaktion zwischen den Klassen wird dann in Abschnitt 4.4 mittels eines Sequenzdiagramms erklärt. Begonnen werden soll mit den Klassen außerhalb des Rahmens. Diese existierten schon vorher, und werden angepasst oder auch nur angesprochen.

- AdserverApp: Die Klasse AdserverApp behandelt alle Aufgaben, die während Start und Herunterfahren des Adservers auftreten. Hier werden auch die Geschäftsobjekte von der Datenbank geladen. Entsprechend ist hier auch anzusetzen, um die Daten stattdessen aus dem Shared Memory zu lesen und diese dorthin zu schreiben.

- Serializer: Jedes Geschäftsobjekt verfügt über einen eigenen Serializer, der das Geschäftsobjekt aus der Datenbankabfrage generiert. Für Filter z. B. gibt es rund 20 unterschiedliche Serializer, deren Basisklasse „FilterSerializer“ heißt. Daher steht die Klasse Serializer im Diagramm für rund 30 bis 40 tatsächliche Klassen, die der Übersichtlichkeit halber auf eine Klasse reduziert wurden. Die zusätzlich integrierte Funktionalität besteht daran, dass die Serializer nun auch dazu in der Lage sein sollen, Daten mittels Thrift zu serialisieren und zu deserialisieren.

- Container: Die dargestellte Klasse Container ist Platzhalter für drei unterschiedliche Klassen. Jeder Grundtyp eines Geschäftsobjekt (also Filter, Banner und Werbeplatz) hat seine eigene Containerklasse, in der alle Objekte in einer Map gespeichert werden. Beim Schreiben ins Shared Memory wird diese durch Iteration ausgelesen, und beim Lesen vom Shared Memory wieder gefüllt. Das enthaltene Objekt in dieser Map ist vom Typ IStorableElement (bzw. ein Zeiger auf dieses Objekt).

- IStorableElement: Ist die grundlegendste Basisklasse aller Geschäftsobjekte.

Als nächstes werden die Klassen im Rahmen beschrieben, die eigens entwickelt werden sollen, um die Kernfunktionalitäten zu ermöglichen:

- BusinessObjectsWriteManager: Diese Klasse gibt die Möglichkeit, die Geschäftsobjekte in unterschiedlichen Formaten zu sichern. So wäre es hier auch möglich, die Objekte nicht ins Shared Memory zu schreiben, sondern auf die Festplatte, falls das später gewünscht sein sollte. Die Klasse ist nicht zwingend notwendig, erhöht aber die Flexibilität bei späteren Anpassungen. Implementiert wird aber nur die Funktionalität für das Schreiben der Geschäftsobjekte ins Shared Memory.
- BusinessObjectsLoadManager: Das Gegenstück zur zuvor genannten Klasse. BusinessObjectsLoadManager kann die Geschäftsobjekte aus unterschiedlichen Quellen lesen. Es wird versucht, die Daten aus dem Shared Memory zu lesen, schläft das fehl, wird die ursprüngliche Methode verwendet, die Objekte aus der Datenbank zu laden.
- SharedMemorySegmentManager: Die zentrale Klasse der geplanten Implementierung. Hier werden alle Shared Memory Segmente verwaltet. Dazu gehört das Lesen und Schreiben in diese, wie in Abschnitt 4.2 beschrieben, und die Ansprache der Serialisierungsmethode in der ebenfalls zuvor genannten Klasse Serializer.
- BusinessObjectsTypeMapper: Es gibt unterschiedliche Serializer, die das eigentliche Objekt erstellen. Jedes Geschäftsobjekt verfügt über einen zugeordneten Objekttypen (z. B. Bundeslandfilter oder Betriebssystemfilter), der eine Klassenvariable der Basisklasse IStorableElement ist. Dieser Objekttyp ist vom Typ String und dient unter anderem dazu, den korrekten Serializer anzusprechen. Nur der richtige Serializer kann das Objekt korrekt erstellen. Ein String ist allerdings nicht speichereffizient, da er mehr Platz benötigt, als notwendig ist. Daher wird im Shared Memory der Typ mit dem Datentyp Integer gespeichert (genauer uint32_t). Der BusinessObjectsTypeMapper hat nun die Aufgabe, den Integer in einen String umzuwandeln und umgekehrt, damit immer mit dem Datentyp gearbeitet werde kann, der erwartet wird.
- ThriftClasses: Dies sind die von Thrift automatisch generierten Klassen, die zur Serialisierung und Deserialisierung verwendet werden.
- duration_statistics: Ist ein Element des SharedMemorySegmentManagers, und dient nur dazu, statistische Daten während des Startens zu sammeln (z. B. Dauer die für die Objekterstellung benötigt wird).

Die Interaktion der Klassen soll nun im nachfolgenden Kapitel erklärt werden.

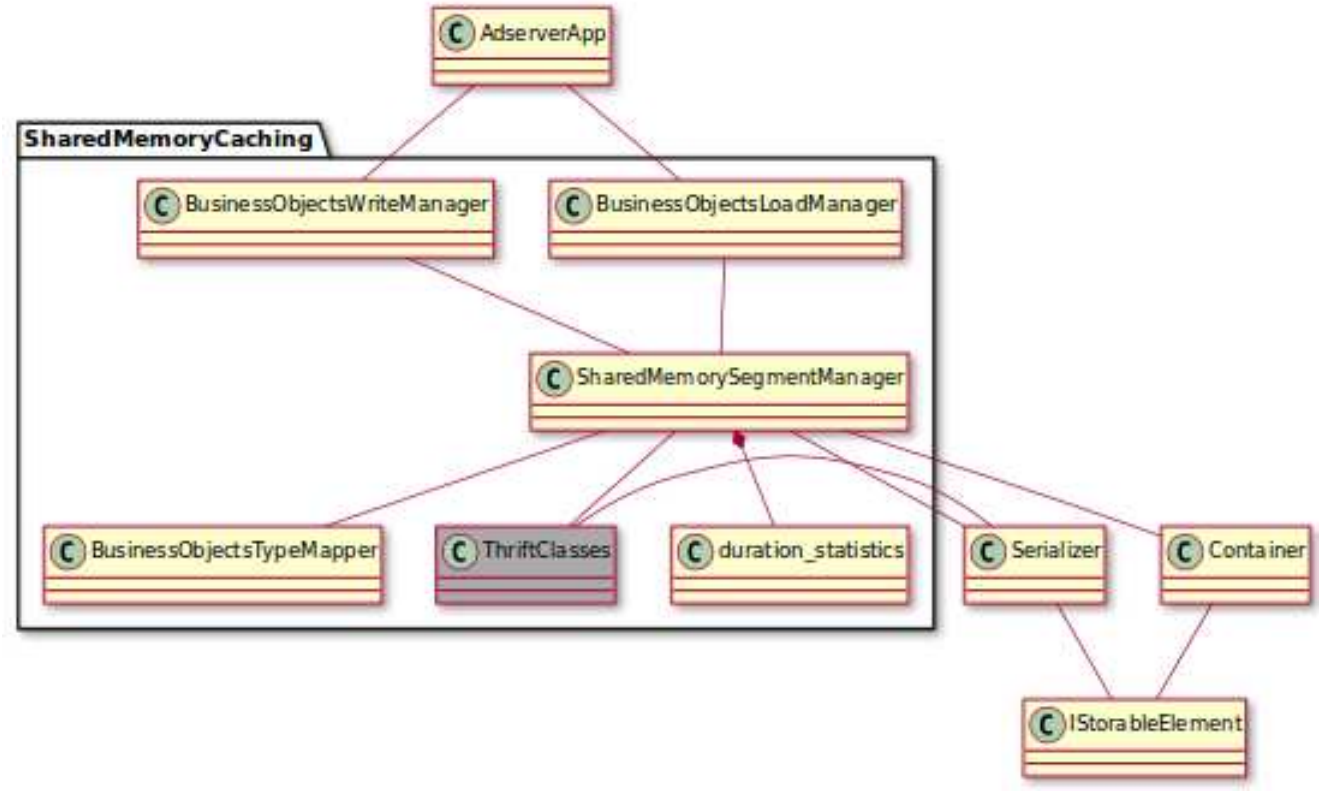

Abbildung 4.4.: Klassendiagramm der geplanten Implementierung. Die Klassen im Rahmen wurden neu entwickelt. Alle übrigen wurden modifiziert oder werden verwendet. Der Übersicht halber fehlen Methoden und Variablen.

4.4. Sequenzdiagramm

Wie in Abschnitt 3.3.2 beschrieben soll Shared Memory wie folgt in den Adserver integriert werden:

1. Beim ersten Start werden die Objekte von der Datenbank geladen.
2. Während des Herunterfahrens werden diese ins Shared Memory geschrieben.
3. Beim Neustart sind die Daten im Shared Memory vorhanden, daher wird auch aus diesem geladen.

Chronologisch werden die Neuanpassungen also als erstes beim Herunterfahren des Adservers aufgerufen. Abbildung 4.5 beschreibt (leicht vereinfacht), die Interaktion der einzelnen Komponenten, um dies zu ermöglichen. Dies wird nachfolgend beschrieben:

1. Ein Benutzer oder ein anderes Programm gibt dem Adserver das Signal zum

Herunterfahren. Infolgedessen wird die finalize()-Methode in der Klasse AdserverApp aufgerufen.

2. In dieser wird ein Objekt vom Typ BusinessObjectsWriteManager angelegt.
3. BusinessObjectsWriteManager wiederum erzeugt ein Objekt vom Typ SharedMemorySegmentManager.
4. Die Klasse AdserverApp ruft nun die Methode writeBusinessObjects vom BusinessObjectsWriteManager auf.
5. Innerhalb dieser Methode wird die Methode writeToSharedMemory aufgerufen, die in derselben Klasse vorhanden ist.
6. Jetzt wird die Methode writeObjectsToSharedMemory aufgerufen, die in der Klasse SharedMemorySegmentManager liegt.
7. Die dann aufgerufene Methode writeBusinessObjectsToSharedMemory() ist ein Platzhalter für drei Methoden, für jeden Objekttypen einen (Filter, Banner, Werbeplatz). Diese drei Methoden machen das gleiche für ihren jeweiligen Objekttypen, insofern ist es hier vereinfacht dargestellt.
8. Nun wird die Methode getDataMap des Containers aufgerufen. Da jeder Objekttyp auch einen eigenen Container hat, wird dieser Schritt für jeden Container wiederholt (Siehe Rahmen in der Abbildung).
9. Diese liefert die DataMap zurück. Es handelt sich dabei um die Map aller Geschäftsobjekte.
10. Über diese Map wird iteriert, und jedes Objekt wird über die Methode serializeToThrift des passenden Serializers serialisiert.
11. Rückgabe ist das serialisierte Objekt als String.
12. Die Methode writeSegment wird aufgerufen, wenn genügend Objekte serialisiert worden sind, um ein 32 MB Segment zu füllen. Wie in 4.1 beschrieben werden die serialisierten Objekte daher zunächst in einer Liste zwischengespeichert, deren Gesamtgröße fortwährend überprüft wird.
13. Jedes serialisierte Objekt wird gelöscht, um keinen doppelten Speicherplatz zu belegen.
14. Wurden alle Segmente erfolgreich erstellt, wird über die Methode writeMetadata ein Metadaten-Segment geschrieben. Dieses Segment hat einen festen

Schlüssel, und enthält eine Liste der Schlüssel aller übrigen erstellten Segmente, so dass klar ist, welche Segmente beim Start wieder ausgelesen werden müssen. Zusätzlich wird hier auch die Anzahl der Objekte pro Segment abgelegt.

15. Die Methode writeObjectsToSharedMemory liefert schließlich zurück, ob der Vorgang erfolgreich abgeschlossen worden ist. Bricht der aus unbekannten Gründen ab, wird kein Metadaten-Segment geschrieben, entsprechend können die Segmente auch nicht mehr ausgelesen werden (siehe auch nächstes Sequenzdiagramm).

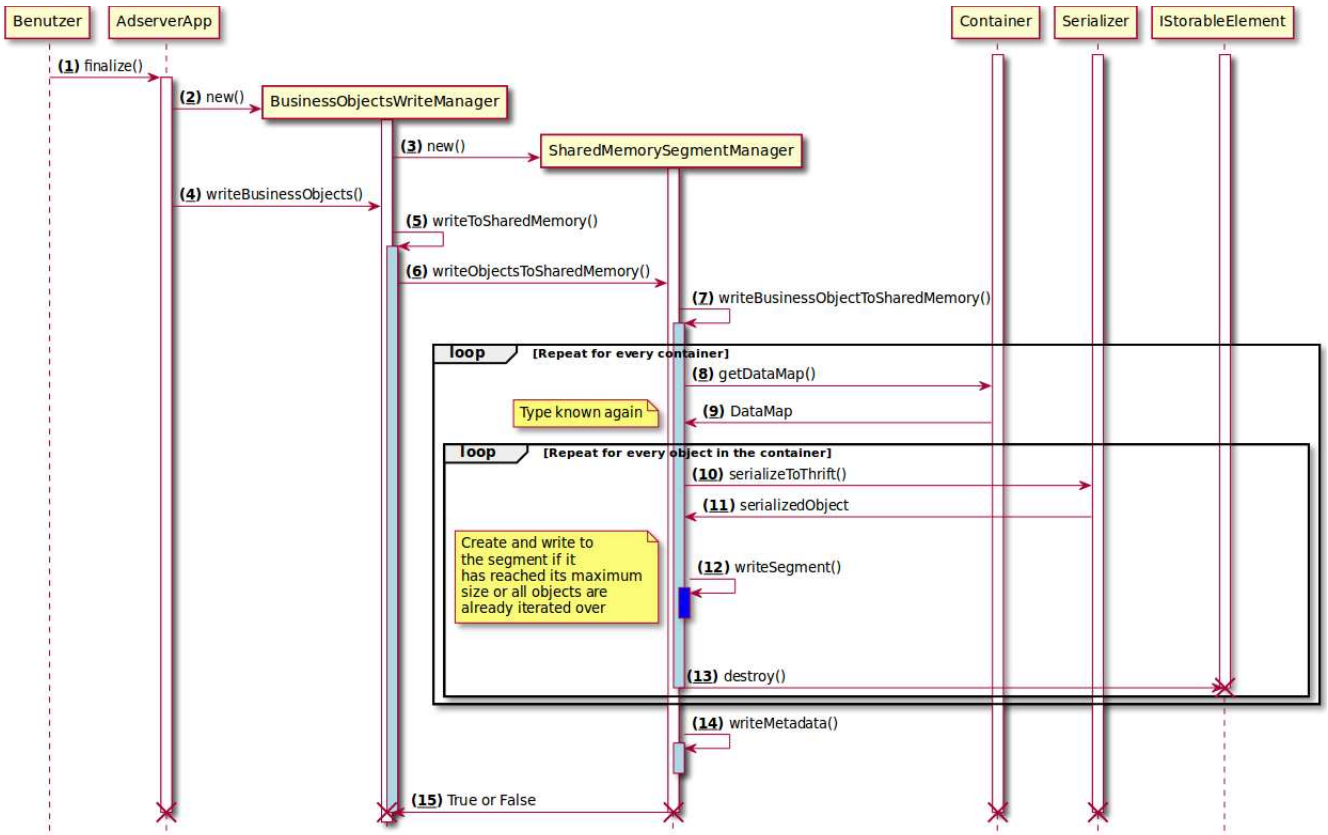

Abbildung 4.5.: Sequenzdiagramm für Schreiboperationen beim Herunterfahren eines Adservers

Im Erfolgsfall stehen nun alle Geschäftsobjekte im Sharded Memory zur verfügen, so dass diese wie in Abbildung 4.6 dargestellt wieder beim Starten des Adservers ausgelesen werden können:

1. Ein Benutzer oder ein anderes Programm startet einen Adserver, der infolgedessen die Methode initialize der Klasse AdserverApp durchläuft.

2. Die Klassen Container...
3. ... und Serializer werden direkt zu Beginn angelegt.
4. Auch eine Instanz der Klasse BusinessObjectsLoadManager wird angelegt.
5. Diese legt ein Objekt vom Typ SharedMemorySegmentManager an.
6. Nun wird die Methode loadBusinessObjects des BusinessObjectsLoadManager aufgerufen, die Daten aus unterschiedlichen Quellen laden kann.
7. Zunächst wird versucht Daten aus dem Shared Memory zu beziehen. Dazu wird die Methode loadFromSharedMemory() aufgerufen.
8. Diese ruft ihrerseits die Methode loadObjectsFromSharedMemory des SharedMemorySegmentManagers auf.
9. Es wird versucht die Metadaten-Informationen auszulesen, die die Liste aller anderen Segmente enthält. Dies geschieht über die Methode readMetadata. Ist kein Metadaten-Segment vorhanden, weil ein Fehler beim Schreiben der Daten aufgetreten ist, wird hier abgebrochen, und stattdessen von der Datenbank geladen.
10. Da durch das Lesen der Metadaten eine Liste aller Segmente zur Verfügung steht, kann nun über diese iteriert werden. Für jedes wird die Methode readBusinessObjectsFromSharedMemory aufgerufen. Diese ist ein Platzhalter für drei andere Methoden, für jeden Objekttypen eine (Filter, Banner und Werbeplätze). Der grundlegende Typ kann über den Nummernkreis der Schlüssel identifiziert werden. Der konkrete Typ muss in den Daten selbst enthalten sein.
11. Innerhalb wird nun die Methode readSegment verwendet um die Daten tatsächlich auszulesen. Diese werden zunächst wieder in eine Liste geschrieben.
12. Ist ein Segment vollständig gelesen, wird es gelöscht, um dessen Speicherplatz freizugeben.
13. Es kann über jedes Objekt im Segment iteriert werden so das mit Hilfe der deserializeFromThrift-Methode das Objekt wieder erstellt werden kann.
14. Diese gibt das Objekt zurück...
15. ..., so dass es wieder in den Container eingefügt werden kann.
16. Ist während des Vorgangs ein Fehler aufgetreten, liefert die Methode loadObjectsFromSharedMemory false zurück.

17. In diesem Falle wird stattdessen von der Datenbank geladen.

Da nun alle Objekte wieder ihren Ursprungszustand haben, kann der Adserver seine Arbeit aufnehmen.

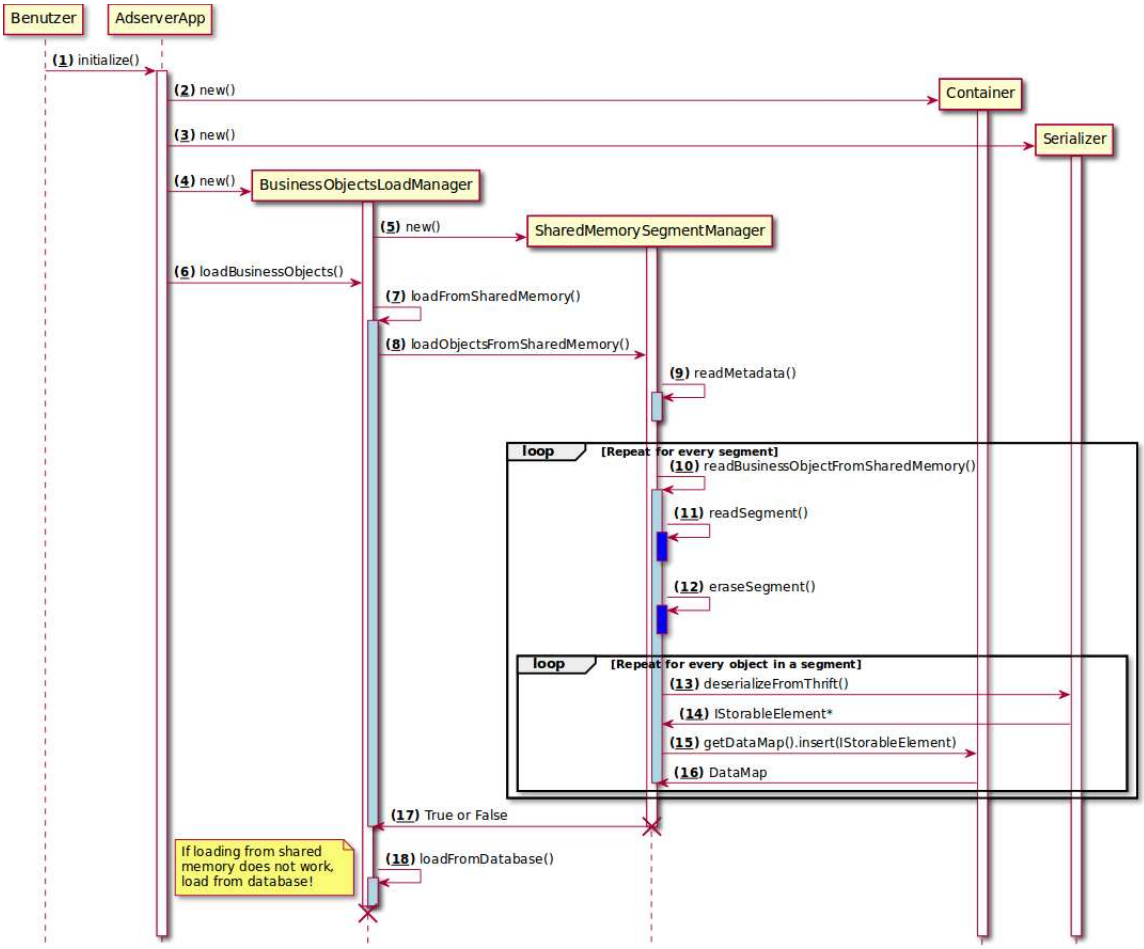

Abbildung 4.6.: Sequenzdiagramm für Leseoperationen beim Starten eines Adservers

4.5. Testdesign

Da die Funktionalität getestet werden sollte, werden Testszenarien benötigt, die die grundlegenden Funktionen auf korrekte Ergebnisse überprüfen soll. Es sind daher mehrere Szenarien zu implementieren:

1. Die Serialisierungs-Methoden selbst müssen getestet werden. Dazu müssen mehrere Objekte der drei Typen Filter, Banner und Werbeplatz erstellt wer-

den, die sich gegenseitig referenzieren. Jedes Objekt ermöglicht bereits standardmäßig das Bilden von Prüfsummen. Die Prüfsumme eines Objekts muss gespeichert werden, dann wird das Objekt serialisiert. Anschließend wird ein neues Objekt aus den erneut deserialisierten Daten erstellt. Wiederum wird die Prüfsumme gebildet und verglichen. Die Werte müssen überein stimmen.

2. Shared Memory Segmente können die im Kernel festgelegte Maximalgröße niemals überschreiten. Diese wird in der Datei „/proc/sys/kernel/shmmax“ festgelegt. Es soll aber möglich sein, diese nach unten hin anzupassen. Daher wird ein Testszenario entwickelt, in dem versucht wird, die Shared Memory Größe über das mögliche Maximum hinaus festzulegen. Das Programm muss so robust reagieren, dass stattdessen das kernelseitige Maximum gewählt wird.

3. Je nachdem, wie groß die Segmengröße gewählt wird, führen die gleichen Daten zu einer unterschiedlichen Menge an Segmenten. Im ersten Szenario wird die Größe so festgelegt, das genau drei solcher Segmente geschrieben werden sollen. Die Anzahl wird verglichen.

4. Im nachfolgenden Szenario wird die Größe so geändert, dass es vier Segmente sein müssen. Auch hier wird wieder verglichen.

5. Zuletzt wird ein Kombinationstest aus lesen und schreiben entwickelt. Shared Memory Segmente werden zunächst geschrieben, und dann wieder ausgelesen über die bereits zur Verfügung stehenden Methoden. Die Menge der Objekte vorher und nachher muss einander gleichen.

5. Implementierung

Für die Implementierung ist es nun nur noch erforderlich, alle Erkenntnisse der Konzeption in ein funktionsfähiges System zu überführen. Nachfolgend soll beschrieben werden, wie vorgegangen wird, und welche zusätzlichen Erkenntnisse während dieser Phase aufgetreten sind.

5.1. Thrift-Definition

Bevor die Geschäftsobjekte ins Shared Memory geschrieben werden können, müssen sie zunächst serialisiert werden. Daher ist der erste Schritt, für jeden Objekttypen eine Thrift-Definition zu erstellen (siehe Abschnitt 4.1.1), die automatisiert Serialisierungs- und Deserialisierungsmethoden erstellt. Bisher wurden die notwendigen Daten aus der Datenbank ausgelesen, um basierend auf diesen die Objekte zu erstellen. Nun hat sich zwar die Datenquelle geändert, weil die Geschäftsobjekte aus dem Shared Memory geladen werden, an den Objekten selbst ändert sich dadurch aber nichts. Insofern kann der Datenbank-Ladevorgang analysiert werden, um herauszufinden, welche Daten benötigt werden, um welches Objekt zu erstellen. Das sind die Daten die im Shared Memory gespeichert werden müssen, und damit auch die Daten, die für die Thrift-Definition zu beachten sind. Verwendet eine Klasse einen Datentypen, der nicht von Thrift unterstützt wird, muss dieser übersetzt werden. Ist eine Klassenvariable z. B. von Typ vector<Filter*>, so taucht dieser als list<int> in der Thrift-Definition auf. Es wird dann statt dem eigentlichen Zeiger, nur die ID des referenzierten Objekts gespeichert. Beim Starten des Adservers kann die ID verwendet werden, um wiederum einen Zeiger zum Objekt zu erhalten. Wie in Abschnitt 4.1 beschrieben, werden die Geschäftsobjekte ja in einer Reihenfolge geladen, die sicherstellt, dass alle referenzierten Objekte bereits vorher vorhanden sind. Der Zeiger wird dann statt des Integers in den Vektor eingefügt. Entsprechende Methoden, die basierend auf der ID einen Zeiger zurückliefern, sind bereits standard-

mäßig verfügbar, und müssen nicht eigens programmiert werden. Nach eingehender Analyse wurden rund 35 unterschiedliche Objekttypen identifiziert, die aber teilweise deutliche Gemeinsamkeiten aufweisen. Daher war es möglich, die Anzahl der Thrift-Klassen auf unter zehn zu reduzieren.

5.2. Implementierung als Skelett hinzufügen

Die Tests (siehe Abschnitt 4.5) werden vor der eigentlichen Implementierung entwickelt. Da diese allerdings Methoden und Klassen ansprechen, die noch nicht vorhanden sind, können die Tests nicht fehlerfrei kompiliert werden. Um dieses Problem zu umgehen, wird nur das Skelett der Klassen (siehe Kapitel 4.3) vorab hinzugefügt. Das heißt, dass nur Methodenrümpfe ohne Funktion vorhanden sind. Da das Klassendiagramm bereits vorher definiert worden ist, kann ein entsprechendes Skelett innerhalb kürzester Zeit integriert werden. Dadurch ist es möglich, die Tests zu entwickeln, ohne auf nicht existente Funktionen zurückgreifen zu müssen. Auch die Thrift-Definition wird vorab benötigt, weshalb sie bereits zuvor erstellt worden ist.

5.3. Test-Implementierung

Wird wie in Abschnitt 5.2 beschrieben vorgegangen, können die Tests bereits direkt nach der Fertigstellung ausgeführt werden. Diese schlagen fehl, da die tatsächliche Funktionalität noch nicht vorhanden ist. Der Vorteil ist jedoch der, dass parallel zum Entwicklungsprozess immer wieder Tests ausgeführt werden können. Es ist zu erwarten, dass diese nach und nach erfolgreich abgeschlossen werden, während immer neue Methoden und Klassen ausgearbeitet werden. So kann kontinuierlich die Qualität überprüft werden. Zum Testen wird das Google C++ Testing Framework verwendet. Das Kernprinzip ist die Formulierung einer Behauptung für einen Testfall, die zu wahr oder falsch evaluiert(Google Inc., 2011). Nachfolgendes Beispiel, ist der simpelste mögliche Test. Es wird überprüft, ob die Zahl Eins gleich der Zahl Eins ist. Als Parameter wird der Name des Tests und der konkrete Testfall mit angegeben:

```
TEST(name_testfall, name_test)
{
    EXPECT_EQ(1, 1);
}
```

Für die in Abschnitt 4.5 beschriebenen Testfälle werden die formulierten Behauptungen ebenfalls auf Korrektheit überprüft. Es wurden wie geplant alle fünf Testfälle implementiert. Werden alle fünf Testfälle erfolgreich abgeschlossen, sind die Kernfunktionen als korrekt zu bezeichnen. Was sich mit den Tests nicht simulieren lässt, ist ein Neustart der Applikation. Zur Verifizierung wird daher zusätzlich ein Adserver installiert, der die neuen Änderungen beinhaltet. Der gleiche Testaufbau wie in Abschnitt 2.3.1.1 beschrieben, wird dann dazu verwendet, um zu überprüfen, ob sich der Adserver wie erwartet verhält. Parallel kann die in Abschnitt 4.2 beschriebene Liste der Shared Memory Segmente eingesehen werden, um die Existenz dieser zu sicherzustellen.

5.4. Änderungen im Adserver

Die Klassen und deren Interaktion sind detailliert in den Abschnitten 4.3 und 4.4 beschrieben. Da es den Rahmen sprengen würde, weitere Details zu erwähnen, soll nachfolgend nur auf die Reihenfolge der Implementierung sowie einige Besonderheiten eingegangen werden, die während der Implementierung zu Tage getreten sind.

Dem in Abschnitt 4.2 beschriebenen Prototyp ist zu entnehmen, dass die Geschäftsobjekte, bevor sie serialisiert werden können, zunächst in die entsprechende Thrift-Klasse kopiert werden müssen. Daher müssen für alle Objekttypen Kopierfunktionen integriert werden, die anschließend automatisch das kopierte Objekt serialisieren bzw. deserialisieren. Um Code-Redundanz zu vermeiden, müssen Objekte, die dieselbe Datenstruktur haben, möglichst zusammengefasst werden. Jeder Objekttyp verfügte bereits zuvor über eine eigene Serialisierungsklasse, die bisher das Objekt aus der Datenbankabfrage generiert hat. Ergänzt werden zusätzliche Methoden, die mittels Thrift das Objekt in eine Zeichenkette serialisieren, bzw. aus einer Zeichenkette das Objekt deserialisieren können. Die serialisierte Zeichenkette kann dann ins Shared Memory kopiert werden, bzw. von dort aus zurück in den eigentlichen Prozess. Da für den Zugriff auf den korrekten Serialisierer der Typ des Objekts be-

kannt sein muss, ergibt sich für jedes Objekt folgende Datenstruktur, die im Shared Memory abgelegt wird:

- Size: Der erste, vier Byte lange Wert, der im Shared Memory abgelegt wird, ist die Größe des Objekts in Bytes.
- Type: Der zweite, ebenfalls vier Byte lange Wert, ist der Typ des Objekts, repräsentiert durch einen Integer.
- Data: Anschließend kommen die Daten selbst.

Diese Struktur wird für jedes Objekt so lange wiederholt, bis ein Shared Memory Segment voll ist (in der Regel bis zu 32 MB). Da die ersten beiden Einträge eine fixe Länge haben, und die Länge des Datenelements ebenfalls bekannt ist, kann durch einen Offset bestimmt werden, an welcher Position das nächste Objekt anfängt, welches wiederum aus Size, Type und Data besteht. So werden nach und nach alle Objekte geschrieben und beim Neustart wieder ausgelesen. Im Metadaten-Segment (siehe Abschnitt 4.4) steht auch die Anzahl der Objekte, so dass bekannt ist, wie viele dieser Wert-Trios auszulesen sind.

Wie in Abschnitt 4.2 beschrieben, benötigt eine Shared Memory Segment eine einzigartige Identifikationsnummer. Aktuell wird für jeden Objekttypen die ID aus einem fixen Nummernkreis gezogen:

- Filter: Nummernkreis beginnt ab 10100.
- Banner: Nummernkreis beginnt ab 20100.
- Werbeplätze: Nummernkreis beginnt ab 30100.
- Metadaten: Es gibt nur ein Medadatensegment mit der Nummer 40100.

Die Entscheidung dafür, feste Nummern statt Zufallszahlen zu wählen, ist dann problematisch, wenn andere Prozesse ebenfalls über Shared Memory kommunizieren, und deren Identifikationsnummern sich mit den gewählten Nummernkreisen überschneiden. Dies ist aber aktuell nicht der Fall. Der Vorteil fester Werte ist der, dass Fehler schneller entdeckt werden können. Tritt der Fall ein, dass Shared Memory Segmente nach dem Auslesen nicht korrekt gelöscht werden, fällt dies spätestens beim Herunterfahren des Adservers auf, da sie erstellt und beschrieben werden sollen, aber bereits vorhanden sind. Wird stattdessen mit Zufallszahlen gearbeitet, werden immer neue Segmente angelegt, die nie gelöscht werden. Nach einigen Zyklen ist der Rechner überlastet, weil kein Speicherplatz mehr vorhanden ist. Wird die in

Abschnitt 4.1 angesprochene Boost-Bibliothek zur Verwaltung der Shared Memory eingesetzt, was mittelfristig geplant ist, entfällt dieses Problem. Diese verwendet zur Identifikation der Segmente Zeichenfolgen statt Zahlen, die bereits anfangs so definiert werden können, dass sie einzigartig sind.

Einer der kritischen Faktoren bei der Deserialisierung der Geschäftsobjekte aus dem Shared Memory ist deren Konsistenz. Wird von der Datenbank geladen, ist es problematisch, wenn Objekte während der Laufzeit des Adservers inkonsistent werden, und damit von denen in der Datebank abweichen. Das kann theoretisch nicht passieren, da Änderungen an der Datenbank dem Adserver direkt mitgeteilt werden, so dass dieser direkt von der Datenbank nachlädt. Sollte der Benachrichtigungsmechanismus aber im Fehlerfalle fehlschlagen, sind die Änderungen in der Datenbank nicht im Adserver abgebildet. Ein Objekt wird inkonsistent. Wird ein Adserver über die Datenbank neu gestartet, werden wieder die korrekten Objekte aus der Datenbank geladen und erzeugt. Das entstandene Problem ist weitaus größer, wenn stattdessen Shared Memory zur Persistierung der Objekte genutzt wird. Wird ein Geschäftsobjekt zur Laufzeit aus unbestimmten Gründen inkonsistent, wird das defekte Objekt im Shared Memory gespeichert, und von dort auch wieder geladen. Es findet unter Umständen wochenlang keine Synchronisation mit dem korrekten Objekt in der Datenbank statt. Abseits davon, dass Geschäftsobjekte theoretisch nicht inkonsistent werden sollten, sind bereits standardmäßig Sicherheitsmechanismen für diesen Fall integriert. Die Konsistenz der Objekte wird fortwährend mittels eines externen, bereits vorhanden Programms geprüft. Dieses vergleicht, ob die Geschäftsobjekte eines Adservers, denen in der Datenbank entsprechen. Dazu wird ein Prüfsummen-Vergleich vorgenommen. Dadurch wird sichergestellt, dass die Geschäftsobjekte, die aus dem Shared Memory deserialisiert worden sind, auch korrekt sind. Etwaige Fehler werden gemeldet. Das Verfahren unterliegt Einschränkungen, die im Kapitel 7.1 erläutert werden.

6. Auswertung

Da jetzt alle geplanten Implementierungen umgesetzt worden sind, muss deren Funktionsfähigkeit und insbesondere der tatsächliche Geschwindigkeitsgewinn gemessen werden. Ersteres wird mit Hilfe der vorab entwickelten Tests validiert. Nach mehreren Zyklen der Programmanpassung wurden diese alle erfolgreich abgeschlossen. Auf letzteres soll im nächsten Abschnitt genauer eingegangen werden.

6.1. Vorher-/Nachher-Vergleich

Zum vergleichen der Startdauer mit und ohne Optimierung wurde derselbe Testaufbau verwendet, wie in Abschnitt 2.3.1.1 beschrieben ist. Allerdings reicht die Verwendung eines Adservers vollkommen aus, da mehrere gleichzeitige Adserver-Starts sich nun nicht mehr negativ beeinflussen können. Die Zeit die ursprünglich für einen Adserver-Start im Testszenario gemessen wurde, betrug 6:17 Minuten. Da inzwischen aber mehrere Monate vergangen sind, wurde zwecks Vergleichbarkeit erneut eine Vorher-Messung durchgeführt. Die Ergebnisse werden in Tabelle 6.1 dargestellt. Die Dauer wurde von ca. fünf Minuten auf zwei Minuten reduziert, was einem Geschwindigkeitszuwachs von rund 60 % entspricht. Zu beachten ist, dass ein Start mit vergleichbaren Daten auf einem Live-System aktuell rund 900 Sekunden benötigt (siehe auch Abschnitt 2.1), statt der dargestellten 301 Sekunden auf dem Testsystem, da bei einem Live-System z. B. die Datenbank mit zusätzlichen Aufgaben betreut ist. Wird z. B. die Replikation der Datenbank auf der Testdatenbank aktiviert (siehe 2.3.1.1 und Abbildung 2.3) dauert der Start rund 1800 Sekunden. Die optimierte Variante benötigt aber weiterhin 119 Sekunden. Daher ist zu erwarten, dass das Potential der Optimierung höher ausfällt, als die Daten vermuten lassen. Zusätzlich wurde die Dauer für das Laden der Kampagnen mit ausgegeben. Wie in 2.1 bereits beschrieben, werden diese asynchron geladen. Damit stehen Kampagnen erst einige Zeit nachdem der Adserver beginnt Werbung auszuliefern zur Verfügung.

So kann es dazu kommen, das ein Adserver für die Dauer, die benötigt wird um alle Kampagnen zu laden, Werbung falsch ausliefert. Dieser Teil wurde nicht optimiert, die Dauer fällt daher in der Vorher- und der Nacher-Variante an. Der Wert ist angegeben, um zu evaluieren, ob es sich lohnt, die Kampagnen mit in die Optimierung mit aufzunehmen (siehe Kapitel Ausblick).

Dauer (unoptimiert)	Dauer (optimiert)	Laden der Kampagnen (asynchron)
~ 301 Sekunden	~ 119 Sekunden	~ 50 Sekunden

Tabelle 6.1.: Vorher/Nachher-Vergleich der Startdauer eines Adservers.

Aus den Beschreibungen aus Abschnitt 4.3 ist ersichtlich, dass automatisiert Statistiken während des Starts erstellt werden, mit der sich zusätzliche Informationen darstellen lassen. Diese sind in Tabelle 6.2 dargestellt, und betreffen nur Zusatzinformationen zu Bannern. Zusammenfassend ist die zentrale Information, dass nun fast die gesamte Zeit mit der Objekterstellung selbst verbracht wird. Um die Objekterstellung zu messen, wurde ein Timer um den Konstruktor des Objekts gelegt. Hier ist anzumerken, dass die Daten aus dem Thrift-Objekt ins eigentliche Objekt kopiert werden müssen. Eine vertretbare Vermutung wäre die Behauptung, der Kopiervorgang wäre zeitaufwendig. Allerdings wird beim Herunterfahren des Adservers für das Schreiben eines Shared Memory Segments mit der Größe 32 MB weit weniger als eine Sekunde benötigt. Der Kopiervorgang beim Herunterfahren ist der gleiche, wie der Kopiervorgang beim Starten. Was bei ersteren jedoch nicht in die Zeit mit einfließt, ist das Erstellen des Objekts, da dies nur beim Start notwendig ist. Insofern ist die Objekterstellung exklusive des Kopiervorgangs das Problem. Im Konstruktor der Banner muss daher der Aufruf ein oder mehrerer Methoden stattfinden, die viel Zeit in Anspruch nehmen. Die Befürchtung, dass das Serialisieren der Objekte aufwendig ist, lässt sich daher ebenso wenig bestätigen (siehe Abschnitt 4.2.1), wie die Annahme, das Kopieren der Objekte würde lange dauern. Die Objekterstellung soll im folgenden Abschnitt näher betrachtet werden.

6.2. Optimierung der Objekterstellung

Die in Abschnitt 2.4 angesprochenen Hauptprobleme, die durch die Kommunikation mit der Datenbank über das Netzwerk entstanden sind, wurden behoben. Da dieser Teil beim Neustart eines Adservers komplett entfällt, wird nun ein zunächst verbor-

Dauer je 32 MB Segment	Anzahl Segmente	Total	Dauer Deserialisierung	Dauer Objekterstellung
~ 5 Sekunden	21	105 Sekunden	3 Sekunden	100 Sekunden

Tabelle 6.2.: Statistiken für den Start eines optimierten Adservers. Die Statistiken werden nur für Banner geführt, da die beiden übrigen Objektarten einen vernachlässigbaren Anteil an der Gesamtdauer haben. Daher weicht die Dauer in der Spalte „Total“ von der „Dauer (optimiert)“ in Tabelle 6.1 ab.

gener Engpass sichtbar. Das Erstellen der Banner dauert relativ lange, wenngleich nicht so lange wie das Laden über die Datenbank selbst. Die Implementierung des Shared Memory Ansatzes ändert an diesem Problem nichts. Da die Daten serialisiert werden, müssen die Objekte beim Neustart neu angelegt werden (siehe 4.1). Das war in der ursprünglichen Variante auch nicht anders. Abseits des ursprünglichen Ansatzes ist es daher zumindest sinnvoll, den Engpass konkreter zu identifizieren. Wenn eine Optimierung zeitnah möglich ist, kann diese zusätzlich implementiert werden. Um den Engpass auf einige wenige Codestellen einzugrenzen, wurden nach und nach Funktionen abgeschaltet, und der Adserver neu gestartet. Damit ist wiederum ein Vorher/Nachher-Vergleich möglich. Zwar ist ein Banner dann nicht mehr im validen Zustand, dies spielt aber keine Rolle für die Messung während des Starts. Für den Test sind nur zwei Bannertypen von Belang: HTML-Banner und Rawtext-Banner. Diese rufen in ihrem Konstruktor Methoden auf, die wiederum andere Methoden aufrufen. Dies ist bei anderen Bannertypen nicht der Fall. Nachdem mehrfach gezielt Code auskommentiert wurde, lies sich der Engpass auf zwei Funktionsaufrufe zu externen Bibliotheken eingrenzen:

1. boost::replace_all: Diese Funktion macht nichts anderes, als Zeichenketten in einem String durch andere Zeichenketten zu ersetzen. Das ist z. B. dann sinnvoll, wenn Escape-Sequenzen (z. B. \n um einen Zeilenumbruch zu kennzeichnen) für bestimmte Zwecke anders dargestellt werden sollen. Diese Methode benötigt 40 % der Gesamtzeit, die für die Objekterstellung aufgewendet wird.

2. boost::spirit::parse: Banner enthalten eine URL, die mit Platzhaltern für das konkrete Banner versehen sind, so dass das angezeigte Banner dynamisch geändert werden kann. Damit die Platzhalter korrekt ersetzt werden können, wird aus der URL ein sogenannter Parsetree erstellt. Das ist ein Baum, bei der die Blätter und Äste unterschiedliche Typen haben können. Der Zweck Bannercode zu parsen ist der, zum Zeitpunkt der Auslieferung die Ersetzung

von Platzhaltern in kürzester Zeit ausführen zu können. Ansonsten müsste, wie bei der Methode replace_all, immer durch das gesamte Banner iteriert werden. Durch das Parsen hat man einen Baum mit Zeigern zu den konkreten Elementen des Baums, und den zu durchgehen ist wesentlich schneller. Diese Methode benötigt 50 % der Gesamtzeit, die für die Objekterstellung aufgewendet wird.

Zu ersterem gibt es viele alternative Ansätze, die leicht zu implementieren sind und auf die Verwendung der Boost-Bibliotheken verzichten. Mehrere Varianten wurden ausprobiert, und die schnellste davon, hat die ursprüngliche Methode ersetzt. Damit reduziert sich die Gesamtdauer auf rund 75 Sekunden, statt der 119 Sekunden, die in Tabelle 6.1 angegeben sind. Das Erstellen des Parsetrees hingegen kann nicht ohne einigen Aufwand optimiert werden. Einige Ansätze sind im Kapitel Ausblick beschrieben.

6.3. Fazit der Auswertung

Zusammenfassend ist die Optimierung als erfolgreich zu bewerten, da ein deutlicher Geschwindigkeitszuwachs zu verzeichnen ist. Die in Abschnitt 4.2 geschätze Dauer, die nach der Optimierung voraussichtlich noch für den Adserver-Start benötigt wird, wurde nicht überschritten.

7. Ausblick

Optimierung ist ein Vorgang, in den beliebig viel Zeit investiert werden kann, da sich immer etwas finden lässt, was mit ein paar Anpassungen noch schneller oder effizienter laufen könnte. Im Rahmen der Bachelor-Thesis wurde hauptsächlich nur ein Ansatz verfolgt, der den meisten Zeitgewinn verspricht. Daneben gibt es aber noch viele weitere Möglichkeiten, den Adserver-Start zu beschleunigen. Dazu kommt, dass auch die implementiere Lösung verbessert werden könnte, wenn einige zusätzliche Punkte beachtet werden. In diesem Kapitel soll ein Ausblick gegeben werden, der beschreibt, welche Möglichkeiten noch vorhanden sind, bestehendes zu verbessern, oder neue Wege in der Optimierung zu gehen.

7.1. Konsistenzsicherstellung der Objekte

Das Hauptproblem des gewählten Ansatzes ist sicher die Tatsache, dass die Daten nicht im Shared Memory persistiert werden können, wenn ein Adserver zur Laufzeit abstürzt. Bisherige Analysen zeigen (siehe Kapitel 3), dass dies verhältnismäßig selten vorkommt, und somit nicht im Hauptfokus der Betrachtung liegt. Dazu kommt, dass dieses Problem mit der gewählten Implementierung nicht behoben werden kann, da die aktuelle technische Funktionsweise keine Möglichkeit zulässt, mit Abstürzen umzugehen. Soll dieses Problem abgemildert werden, müssen daher andere Stellen betrachtet werden, z. B. eine Optimierung des Ladens der Daten über die Datenbank. Dies wird weiter unten beschrieben. Die Verwendung des Shared Memory hat aber auch einige andere Nachteile.

Bisher ist davon ausgegangen worden, dass ein Adserver sofort neu gestartet wird, nachdem er heruntergefahren worden ist. In diesem Falle vergehen nur einige wenige Minuten, in denen der Adserver nicht aktiv ist. Alle in dieser Zeit eingehenden Än-

derungen, werden in der Message-Queue zwischengespeichert, so dass der Adserver keine Daten verliert (siehe Abschnitt 3.3.2). Die Annahme der Adserver würde sofort neu gestartet werden, ist eine logische Standardannahme. Was passiert aber, wenn dies nicht der Fall ist, unter der Adserver aus bestimmten Gründen mehrere Stunden oder sogar Tage inaktiv bleibt. In diesem Falle reicht die Message-Queue nicht aus, die Information, welche Daten verändert wurden, zwischenzuspeichern, da sie ab einem bestimmten Zeitpunkt voll ist. Jetzt sind die Daten im Shared Memory veraltet, und auch die nachgeladenen Daten sind nicht mehr aktuell. In diesem Szenario müssen die Daten zwangsläufig von der Datenbank nachgeladen werden. Aktuell ist kein Mechanismus, der dies erkennt und entsprechend reagiert, implementiert. Dies zu ergänzen wäre aber kein Problem. Das Metadaten-Segment kann um einen Zeitstempel ergänzt werden (siehe Abschnitt 4.4), der ausgelesen wird, wenn der Adserver neu startet. Ein Vergleich zwischen dem Zeitstempel der Daten im Shared Memory und der aktuellen Systemzeit genügt dann, um das Alter der Daten zu bestimmen. Dazu muss nur ein Schwellenwert festgelegt werden, ab wann die Daten als zu alt gelten. In diesem Fall wird normal von der Datenbank geladen.

Ein weiteres Problem ist die Verlässlichkeit der im Shared Memory abgelegten Daten. Es wäre im Fehlerfalle denkbar, dass die deserialisierten Objekte nicht den Daten in der Datenbank entsprechen. Ein Mechanismus um diese Situation zu identifizieren, wurde nicht eingebaut, da bereits die Prüfsummen der Objekte aller Adserver auf Konsistenz überprüft werden (siehe Kapitel 5.4). Diese Überprüfung ist auch ausreichend, da somit fehlerhafte Objekte entdeckt werden, dass Problem ist, dass diese Prüfung nur alle 15 Minuten ausgeführt wird. Fällt ein Adserver-Start in ein Zeitfenster kurz nach der Prüfung, und lädt fehlerhafte Daten aus dem Shared Memory, kann es daher im Extremfall dazu kommen, dass bis zu 15 Minuten lang, falsche Werbung ausgeliefert wird. Ein Szenario des es zu vermeiden gilt. Im Normalfall sollte es überhaupt nicht dazu kommen, dass die Objekte fehlerhaft sind, und eine zeitverzögerte Überprüfung findet bereits statt, insofern ist dieses Problem nicht als gravierend anzusehen. Es sollte aber festgehalten werden, dass es existiert. Gegebenenfalls könnten dann zusätzliche Sicherungen implementiert werden. Etwaige Pläne hierzu existieren aber noch nicht.

7.2. Optimierung der Objekterstellung

Durch das lokale Caching der Geschäftsobjekte sind die ursprünglich identifizierten Engpässe behoben worden, und die Objekterstellung tritt nun in den Fokus. Wie bereits in Abschnitt 6.2 beschrieben, wurde eine langsame Methode bereits optimiert. Es verbleibt die Methode boost::spirit::parse() als langsamstes Element beim Adserver-Start. Für die gesamte Objekterstellung von Bannern wurde die Dauer mit 100 Sekunden beziffert (siehe Tabelle 6.2), wobei 50 Sekunden dabei allein auf diese Methode entfallen. Hier ist also noch einiges an Potential vorhanden. Ansätze hier zu optimieren sind aber nicht trivial. Ein Parser wird zwangsläufig benötigt. Zwar stellt sich heraus, dass die zum Parsen verwendete Bibliothek langsam ist, aber einen eigenen Parser zu programmieren ist sehr zeitaufwendig. Um hier Optimierung zu betreiben, sind unterschiedliche Ansätze denkbar:

1. Parsen der Strings erst dann, wenn dieser tatsächlich benötigt wird: Hier wird das Problem im Prinzip nur verschoben. Statt direkt beim Start den Parsetree zu erzeugen, geschieht dies erst zur Laufzeit. Dieses Vorgehen ist nicht als Ideal zu betrachten, da Verzögerungen zur Laufzeit kritischer sind, als Verzögerungen während des Starts.

2. Parallelisierung des Parse-Vorgangs: Aktuell werden alle Strings seriell, also nacheinander, geparsed. Denkbar wäre es, diesen Vorgang zu parallelisieren, und die Berechnung auf mehrere CPU-Kerne aufzuteilen. Dies setzt voraus, das die Geschwindigkeit der Parse-Methode CPU-seitig limitiert ist. Eine Messung mit dstat (siehe Abschnitt 2.3.1.1) hat dies bestätigt. Ein CPU-Kern war während des Parsens beständig am Limit. Insofern ist diese Methode vielversprechend.

3. Serialisieren der Parsestrukturen: Aktuell werden nur die Strings serialisiert und im Shared Memory persistiert. Es wäre denkbar, zusätzlich auch den gesamten Parsetree zu serialisieren, und zu speichern, so dass dieser beim Neustart nicht neu erstellt werden muss. Apache Thrift unterstützt aber derartige Datenstrukturen nicht standardmäßig, so dass hier einiges an Recherche und Einarbeitung notwendig ist, um Wege zu finden.

4. Optimierung des Parsens durch Verwendung einer neuen Bibliothek: Ein weiterer guter Ansatzpunkt ist die Beschleunigung des Parsens selbst, durch die Verwendung einer besseren Bibliothek, bzw. einer neueren Version. Bei ADITI-

ON wird die Boost-Bibliothek in der Version 1.34.1 verwendet. Diese enthält auch die Methoden zur Erstellung von Parsetrees. Die aktuellste Version ist jedoch 1.57.0. Zwischen beiden Versionen liegen mehrere Jahre, und insbesondere die Erstellung der Parsetrees wurde deutlich überarbeitet. Vorab-Tests könnten klären, ob die neuere Version einen deutlichen Geschwindigkeitsschub bringt. Zur Zeit ist ein Update der Version zwar nicht möglich, da bestehende Abhängigkeiten dies verhindern, entsprechende Änderungen am Programmcode sind aber bereits in Arbeit, so dass in absehbarer Zeit ein Wechsel möglich sein sollte.

Von den genannten Methoden ist die Parallelisierung des Parsens oder ein Update der Boost-Version das beste Vorgehen, und könnten nochmals zu einem deutlichen Geschwindigkeitsgewinn führen.

7.3. Optimierung der Datenbankabfragen

Wie in Abschnitt 7.1 erklärt, ist im Falle eines Absturzes immer von der Datenbank zu laden. Da an diesem Prozess nichts optimiert wurde, dauert der Start so lange wie zuvor[1]. Als eine der Hauptfaktoren hierfür war die Vielzahl einzelner SQL-Abfragen genannt, die zu vermehrter Latenz in der Netzwerkkommunikation führen. Bisher war es so, dass auf alle Hauptobjekte (wie z. B. Banner) in einer Abfrage zugegriffen worden ist, deren Unterobjekte aber alle einzeln nachgeladen werden. Gibt es über 100.000 Banner, jeder mit nur einem zugeordneten Filter, führt das zur gleichen Anzahl zusätzlicher Anfragen. Besser wäre es, auch diese über einen einzigen Datenbankzugriff anzufordern. Die Ergebnismenge kann dann im Hauptspeicher gehalten werden, um die Geschäftsobjekte aus diesem zu erstellen. Dieser Lösungsansatz weist Parallelen zur Verwendung des Shared Memories auf. Die Netzwerkkommunikation wird verringert, stattdessen wird der Arbeitsspeicher intensiver genutzt, um den Startprozess zu beschleunigen. Entsprechende Änderungen am Adserver-Code sollten technisch wenig aufwendig sein, so dass hier ein weiterer guter Ansatzpunkt vorhanden ist, um den normalen Start mit Datenbankzugriff zu beschleunigen.

[1] Abgesehen von der Optimierung in der Objekterstellung, der zu einer Beschleunigung führt

7.4. Optimierung der Kommunikation mit dem XML-Daemon

Änderungen von Geschäftsobjekten werden dem Adserver über den XML-Daemon mitgeteilt (siehe Abschnitt 3.3.1). Entsprechende XML-Nachrichten enthalten die Identifikationsnummer des Objekts, dessen Typen, und die Art der Änderungen. Die eigentliche Änderung selbst, wird aber nicht mitgeteilt. Mit diesen Informationen ist es dem Adserver nun möglich, das komplette Objekt aus der Datenbank nachzuladen. Derartige Nachrichten treffen im Adserver sehr häufig ein, was dazu führt, dass der Verarbeitungsaufwand zeitintensiv ist. Dies ist der Grund, aus dem die Serialisierung der Geschäftsobjekte in eine Datei nicht in Frage kommt (ebenfalls Abschnitt 3.3.1). Dieser Ansatz hat jedoch den Vorteil, das Abstürze des Adservers keine Auswirkung auf die lokale Speicherung der Geschäftsobjekte hat. Soll daher dieser Weg weiter verfolgt werden, ist es erforderlich, die Art wie der XML-Daemon Änderungen an den Adserver kommuniziert, zu verändern. Folgende Änderungen wären denkbar, um den Aufwand für die Änderung von Objekten zur Laufzeit zu minimieren:

- Kleine Änderungen werden direkt in der XML-Nachricht mitgeschickt. Da die Nachricht sowieso geparsed werden muss, ist der Aufwand um zusätzliche Informationen zu extrahieren, vermutlich gering. Es kann dann gezielt ein einzelnes Attribut des Objekts im Arbeitsspeicher geändert werden, ohne gleich das komplette Objekt aus der Datenbank nachladen zu müssen. Dadurch sollte der Zeitaufwand der Objektveränderung geringer werden.
- Ebenfalls denkbar wäre es, nur einzigartige Objektänderungen über den XML-Daemon zu verschicken. Aktuell kann es passieren, dass dasselbe Objekt im Abstand von zehn Sekunden geändert wird. Beide Änderungen werden dem Adserver bekanntgegeben, und beide Male wird das Objekt nachgeladen. Da aber die letzte Änderung die erste sozusagen überschreibt, reicht es aus, das Objekt nur einmal nachzuladen, und der Aufwand halbiert sich. Um das Verhalten zu ändern, müssten die Message-Queues dahingehend geändert werden, nur einzigartige Einträge zu enthalten. Dazu ist es erforderlich bei jeder neuen Änderung zu überprüfen, ob eine Änderungen zum selben Objekt bereits in der Warteschlange ist. Das kostet ebenfalls Zeit, daher sollte zuerst evaluiert werden, ob dieses Vorgehen tatsächlich sinnvoll ist.

7.5. Fazit zum Ausblick

Wie zu sehen ist, gibt es viele mögliche Ansatzpunkte, um weitere Optimierung zu betreiben. Die hier aufgelisteten Vorschläge, behandeln auch nur die offensichtlichsten Möglichkeiten. Nichtsdestoweniger ist es gelungen, mit den bereits implementieren Änderungen, einen deutlichen Geschwindigkeitsgewinn zu erreichen, daher sind nicht alle Methoden den Zeitaufwand wert. Das sollte intern abgeschätzt werden.

Zusammenfassung

Die Analyse des Problems wurde eigenständig mit den seitens ADITION zur Verfügung gestellten Möglichkeiten durchgeführt. Lösungsansätze sind mitunter durch eigene Vorschläge entstanden, größtenteils stammen diese jedoch von Mitarbeitern. Die Recherche und Bewertung der Alternativen beruht jedoch auf Eigenleistung. Gleiches gilt für die Konzeption und Implementierung des gewählten Verfahrens. Der Vorher/Nachher-Vergleich wurde ebenfalls selbstständig durchgeführt. Bei der Optimierung der Objekterstellung, die in Kapitel 6.2 beschrieben wird, wurden unterschiedliche alternative Verfahren getestet, und bewertet. Daran involviert waren, neben mir, noch weitere Mitarbeiter. Einige der in Kapitel 7 beschriebenen zusätzlichen Optimierungsmöglichkeiten wurde durch Mitarbeiter angeregt. Meine Eigenleistung besteht hier aus dem Zusammentragen und Ergänzen dieser Verfahren. Zusammenfassend wurde daher das Projekt selbstständig durchgeführt, wobei Erfahrungen und Anregungen von Mitarbeitern stets eingeflossen sind.

A. Anhang

A.1. Thrift-generierte Klasse „PersonThrift“

```
class PersonThrift {
public:
    static const char* ascii_fingerprint;
    static const uint8_t binary_fingerprint[16];

    PersonThrift() : vorName(""), nachName("") {
    }

    virtual ~PersonThrift() throw() {}

    std::string vorName;
    std::string nachName;

    bool operator == (const PersonThrift & rhs) const
    {
        if (!(vorName == rhs.vorName))
        return false;

        if (!(nachName == rhs.nachName))
        return false;

        return true;
    }
    bool operator != (const PersonThrift &rhs) const
```

```
    {
        return !(*this == rhs);
    }
    bool operator < (const PersonThrift & ) const;

    uint32_t read(
        ::apache::thrift::protocol::TProtocol* iprot
    );

    uint32_t write(
        ::apache::thrift::protocol::TProtocol* oprot
    ) const;
};
```

A.2. Prototyp

```
#include "sys/shm.h"
#include "stdio.h"
#include "test_types.h"
#include "test_constants.h"
#include "thrift/protocol/TBinaryProtocol.h"
#include "thrift/transport/TBufferTransports.h"
#include <string>
#include <vector>
#include "Person.h"

using namespace apache::thrift;

bool PersonThrift::operator<(PersonThrift const&) const
{
    return true;
}

bool CarThrift::operator<(CarThrift const&) const
{
```

```
    return true;
}

int main(int argc, char** argv)
{
    int shared_memory_id;
    // Objekt im Speicherbereich des
    // eigenen Prozesses anlegen anlegen
    Person* pers1 = new Person("Dieter", "Schmidt");

    // Objekt in Thrift-Definition kopieren
    boost::shared_ptr<transport::TMemoryBuffer> transport(
        new transport::TMemoryBuffer
    );
    boost::shared_ptr<protocol::TBinaryProtocol> protocol(
        protocol::TBinaryProtocol(transport)
    );

    PersonThrift personThrift;
    personThrift.nachName = pers1->getNachname();
    personThrift.vorName = pers1->getVorname();
    // ---------

    // Objekt serialisieren
    personThrift.write(protocol.get());

    uint8_t* pbuf;
    uint32_t size;
    // Daten in pbuf schreiben und Größe
    // der Daten in size schreiben
    transport.get()->getBuffer(&pbuf, &size);
    transport->flush();
    transport->close();

    char* shared_memory_start_pointer;
```

```
    // Shared Memory Segment anlegen über shmget,
    // shared_memory_id als Identifikator
    if ((shared_memory_id = shmget(1000,
        size + sizeof(uint32_t),
        IPC_CREAT | 0666)) != -1)
    {
        // Segment verknüpfen über shmat
        shared_memory_start_pointer = static_cast<char*> (
            shmat(shared_memory_id, 0, 0)
        );
        if (shared_memory_start_pointer == (char*) - 1)
        {
            return false;
        }

        // Position festlegen um Daten zu schreiben
        char* sizeRegion = shared_memory_start_pointer;
        char* dataRegion = shared_memory_start_pointer
            + sizeof(uint32_t);

        // Kopieren der Daten ins Shared Memory
        std::memcpy(sizeRegion, &size, sizeof (uint32_t));
        std::memcpy(dataRegion, &pbuf, size);
    }

    // Segment von Prozess lösen und löschen
    shmdt(shared_memory_start_pointer);
    shmctl(shared_memory_id, IPC_RMID, 0);
    delete pers1;
}
```

Literaturverzeichnis

[Ahlers 2007] Ahlers, Ernst: Die richtige Grundlage fürs Heimnetz. In: *c't - Magazin für Computertechnik, S. 120* (2007), Dezember. – URL https://www.wiso-net.de:443/document/CT__2007121707

[Apache Software Foundation 2014] Apache Software Foundation: *Apache Thrift*. https://thrift.apache.org/. 2014. – Internet-Quelle

[Busch 2014] Busch, O.: *Realtime Advertising*, Springer Gabler, Wiesbaden, 2014. – URL http://dx.doi.org/10.1007/978-3-658-05358-1. – ISBN 978-3-658-05358-1

[CDRinfo.pl 2014] CDRinfo.pl: *Seagate Barracuda ST3000DM001 - Test*. http://dyski.cdrinfo.pl/benchmark/Seagate-ST3000DM001/963.html. 2014. – Internet-Quelle

[Downey 2012] Downey, Allen B.: *Statistik-Workshop für Programmierer*, O'Reilly Vlg. GmbH & Co., 5 2012. – URL http://proquest.tech.safaribooksonline.de/book/statistics/9783868998160/9dot-korrelation/id647966?uicode=offenburg. – ISBN 9783868993424

[Gaztanaga 2012] Gaztanaga, Ion: *Sharing memory between processes - 1.56.0*. http://www.boost.org/doc/libs/1_56_0/doc/html/interprocess/sharedmemorybetweenprocesses.html. 2012. – Internet-Quelle

[Google Inc. 2011] Google Inc.: *Getting started with Google C++ Testing Framework*. https://code.google.com/p/googletest/wiki/Primer. 2011. – Internet-Quelle

[Gray 2003] Gray, J.S.: *Interprocess Communications in Linux*, Prentice Hall, 2003. – URL http://proquest.tech.safaribooksonline.de/book/operating-systems-and-server-administration/linux/0130460427. – ISBN 0-13-046042-7

[Hellmann 2013] Hellmann, R.: *Rechnerarchitektur: Einführung in den Aufbau moderner Computer*. S. 17 – 17, Oldenbourg Wissenschaftsverlag, 2013. – URL http://www.degruyter.com/viewbooktoc/product/231457. – ISBN 9783486720037

[Kingston Technology 2014] Kingston Technology: *Best Practices DDR3-1600*. http://www.kingston.com/en/business/server_solutions/best_practices/ddr3_1600. 2014. – Internet-Quelle

[Limoncelli u. a. 2014] Limoncelli, T.A. ; Chalup, S.R. ; Hogan, C.J.: *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*, Addison-Wesley Professional, 2014. – URL http://proquest.tech.safaribooksonline.de/book/operating-systems-and-server-administration/virtualization/9780133478549/chapter-5dot-design-patterns-for-scaling/ch05lev1sec9_html?uicode=offenburg. – ISBN 0-321-94318-X

[Özsu 2007] Özsu, M. T.: *Principles of Distributed Database Systems*. S. 3 – 15, Springer, 2007. – ISBN 978-1-4419-8833-1

[Schmidt 2013] Schmidt, U.: *Professionelle Videotechnik: Grundlagen, Filmtechnik, Fernsehtechnik, Geräte- und Studiotechnik in SD, HD, DI, 3D*. Springer Berlin Heidelberg, 2013 (SpringerLink : Bücher). – URL http://books.google.de/books?id=fpohBAAAQBAJ. – ISBN 9783642389924

[Shimpi 2014] Shimpi, Anand L.: *Samsung SSD 840 Pro (256GB) Review*. http://www.anandtech.com/show/6328/samsung-ssd-840-pro-256gb-review. 2014. – Internet-Quelle

[Stevanovic 2014] Stevanovic, Milan: *Advanced C and C++ Compiling*. Berkely, CA, USA : Apress, 2014. – URL http://proquest.tech.safaribooksonline.de/book/programming/9781430266679/chapter-4-the-impact-of-reusing-concept/sec2_9781430266679_ch04_xhtml?uicode=offenburg. – ISBN 1430266678, 9781430266679

[Stroustrup 2014] Stroustrup, Bjarne: *Programming: Principles and Practice Using C++, Second Edition*, Addison-Wesley Professional, 2014. – URL http://proquest.tech.safaribooksonline.de/book/programming/cplusplus/9780133796759/14dot-graphics-class-design/ch14lev1sec3?uicode=offenburg. – ISBN 0-13-379675-2